# Проектирование веб-API

API позволяет разработчикам выполнять интеграцию с приложением без знания подробностей на уровне кода. Независимо от того, используете ли вы установленные стандарты, такие как REST и OpenAPI, или более новые подходы, такие как GraphQL или gRPC, освоение проектирования API – своего рода суперспособность. Благодаря этому пользоваться вашими веб-сервисами станет легче, и ваши клиенты, как внутренние, так и внешние, это оценят.

«Автор раскрывает тему просто и доступно, рассматривая широкий круг вопросов в удобной для читателя форме».
*Из предисловия Кина Лейна*

«Книга дает ответы на насущные сложные вопросы с помощью простой философии, ничего при этом не утаивая и предлагая фантастическое знакомство с данной темой».
*Бриджер Хауэлл, SoFi.com*

«Отличный путеводитель по RESTful API».
*Шон Смит, Университет штата Пенсильвания*

«Разнообразные теоретические положения сочетаются здесь с практическими примерами».
*Шейн Корнуэлл, XeroOne Systems*

**Темы, затрагиваемые в книге:**

- характеристики правильно разработанного API;
- ориентированные на пользователя и реальные API;
- принцип Secure by design;
- изменение API, его документирование и проверка.

Арно Лоре – архитектор программного обеспечения с большим опытом работы в банковской сфере. В течение 10 лет использует, проектирует и создает API. Он ведет блог под названием API Handyman и создал сайт API Stylebook.

*Книга предназначена для разработчиков с минимальным опытом в создании и использовании API.*

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК "Галактика"
books@alians-kniga.ru

ISBN 978-5-97060-861-6

Проектирование веб-API

# Проектирование веб-API

Арно Лоре

Арно Лоре

# Проектирование веб-API

# *The Design of Web APIs*

**ARNAUD LAURET**

Foreword by Kin Lane

MANNING
Shelter Island

# *Проектирование веб-API*

**Арно Лоре**

Предисловие Кина Лейна

Москва, 2020

**УДК 004.432**
**ББК 32.972.1**
**Л78**

**Л78** **Арно Лоре**
Проектирование веб-API / Пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 440 с.
**ISBN 978-5-97060-861-6**

Книга, написанная с учетом многолетнего опыта автора в разработке API, научит вас, как собирать требования, как найти баланс между техническими и бизнес-целями и как принимать во внимание запросы потребителя. Рассматриваются основные характеристики API, принципы его изменения, документирования и проверки. Эффективные методы разработки проиллюстрированы множеством интересных примеров.

Рассматриваются основные характеристики API, принципы его изменения, документирования и проверки. Эффективные методы разработки проиллюстрированы множеством интересных примеров.

Издание предназначено для разработчиков, обладающих минимальным опытом в создании и использовании API-интерфейсов.

УДК 004.432
ББК 32.972.1

Original English language edition published by Manning Publications USA, USA. Copyright © 2018
Russian-language edition copyright © 2020 by DMK Press. All rights reserved.
Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-5-97060-861-6 (рус.)
ISBN 978-1-61729-510-2 (анг.)
© 2019 by Manning Publications Co.
© Оформление, издание, ДМК Пресс, 2020

# Оглавление


## Часть I. Основы проектирования API ............ 28

1 ■ Что такое проектирование API ............ 29
2 ■ Проектирование API для пользователей ............ 45
3 ■ Проектирование программного интерфейса ............ 75
4 ■ Описание API с помощью формата описания ............ 111

## Часть II. Проектирование практичного API ... 149

5 ■ Проектирование простого API ............ 151
6 ■ Проектирование предсказуемого API ............ 183
7 ■ Проектирование лаконичного и хорошо организованного API 213

## Часть III. Контекстное проектирование API ... 233

8 ■ Проектирование безопасного API ............ 235
9 ■ Изменение дизайна API ............ 269
10 ■ Проектирование эффективного API для сети ............ 311
11 ■ Проектирование API в контексте ............ 345
12 ■ Документирование API ............ 381
13 ■ Развитие API ............ 413

# Содержание

*Предисловие* . . . . . 13
*От автора* . . . . . 16
*Благодарности* . . . . . 18
Об этой *книге* . . . . . 20
*Об авторе* . . . . . 24
*Об иллюстрации на обложке* . . . . . 25

**Часть I. Основы проектирование API** . . . . . 28

**1** ***Что такое проектирование API*** . . . . . 29

1.1. Что такое API? . . . . . 30
*1.1.1. API – это веб-интерфейс для программного обеспечения* . . . 30
*1.1.2. API превращают программное обеспечение в детали конструктора LEGO®* . . . . . 32
1.2. Чем важна разработка API . . . . . 35
*1.2.1. Открытый или закрытый API – это интерфейс для других разработчиков* . . . . . 36
*1.2.2 API создается, для того чтобы скрыть реализацию* . . . . . 37
*1.2.3. Страшные последствия плохо спроектированных API* . . . . . 38
1.3. Элементы проектирования API . . . . . 42
*1.3.1 Изучение принципов, выходящих за рамки проектирования программного интерфейса* . . . . . *42*
*1.3.2 Изучение всех аспектов проектирования API* . . . . . 43

**2** ***Проектирование API для пользователей*** . . . . . 45

2.1 Правильная точка зрения для проектирования повседневных пользовательских интерфейсов . . . . . 46
*2.1.1 Когда вы фокусируетесь на том, как все работает, это приводит к возникновению сложных интерфейсов* . . . . . *46*

*2.1.2 Когда вы фокусируетесь на том, что могут делать пользователи, это приводит к появлению простых интерфейсов . . . 48*

2.2 Проектирование интерфейсов программного обеспечения . . . 50

*2.2.1 API как панель управления программным обеспечением . . . 50*

*2.2.2 Ориентация на точку зрения потребителя для создания простых API . . . 51*

2.3 Определение целей API . . . 54

*2.3.1 Отвечая на вопросы «что?» и «как?» . . . 55*

*2.3.2 Определение входных и выходных данных . . . 56*

*2.3.3 Выявляем недостающие цели . . . 58*

*2.3.4 Идентификация всех пользователей . . . 61*

*2.3.5 Использование таблицы целей API . . . 62*

2.4 Избегаем точки зрения поставщика при проектировании API . . . 64

*2.4.1 Как избежать влияния данных . . . 65*

*2.4.2 Как избежать влияния кода и бизнес-логики . . . 67*

*2.4.3 Как избежать влияния архитектуры программного обеспечения . . . 69*

*2.4.4 Как избежать влияния организации, где работают люди . . . 70*

*2.4.5 Определение точки зрения поставщика в таблице целей API . . . 72*

**3** ***Проектирование программного интерфейса*** . . . 75

3.1 Знакомство с REST API . . . 77

*3.1.1 Анализ вызова REST API . . . 77*

*3.1.2 Базовые принципы HTTP . . . 79*

*3.1.3 Базовые принципы REST API . . . 80*

3.2 Перенос целей в REST API . . . 81

*3.2.1 Идентификация ресурсов и их связей с таблицей целей API . . . 82*

*3.2.2 Идентификация действий, их параметров и результатов с помощью таблицы целей API . . . 84*

*3.2.3 Представление ресурсов с помощью путей . . . 86*

*3.2.4 Представление действий с помощью протокола HTTP . . . 88*

*3.2.5 REST API и шпаргалка по HTTP . . . 92*

3.3 Проектирование данных API . . . 94

*3.3.1 Проектирование концепций . . . 94*

*3.3.2 Проектирование ответов от концепций . . . 97*

*3.3.3 Проектирование параметров из концепций или ответов . . . 98*

*3.3.4 Проверка параметров источника данных . . . 99*

*3.3.5 Проектирование других параметров . . . 101*

3.4 Достижение баланса при решении проблем проектирования . . . 101

*3.4.1 Примеры компромисса . . . 101*

*3.4.2 Баланс между удобством для пользователя и соответствием* . . . 103
3.5 Почему REST важен при проектировании любого API . . . 104
*3.5.1 Знакомство с архитектурным стилем REST* . . . 104
*3.5.2 Влияние ограничений REST на проектирование API* . . . 106

**4 *Описание API с помощью формата описания*** . . . 111
4.1 Что такое формат описания API? . . . 112
*4.1.1 Спецификация OpenAPI (OAS)* . . . 113
*4.1.2 Зачем использовать формат описания API?* . . . 115
*4.1.3 Когда использовать формат описания API* . . . 118
4.2 Описание ресурсов и действий API с помощью OAS . . . 119
*4.2.1 Создание документа OAS* . . . 120
*4.2.2 Описание ресурса* . . . 121
*4.2.3 Описание операций в ресурсе* . . . 122
4.3 Описание данных API с помощью OpenAPI и JSON Schema . . . 126
*4.3.1 Описание параметров запроса* . . . 127
*4.3.2 Описание данных с помощью JSON Schema* . . . 130
*4.3.3 Описание ответов* . . . 134
*4.3.4 Описание параметров тела* . . . 137
4.4 Эффективное описание API с помощью OAS . . . 140
*4.4.1 Повторное использование компонентов* . . . 140
*4.4.2 Описание параметров пути* . . . 143

**Часть II. Проектирование практичного API** . . . 149

**5 *Разработка простого API*** . . . 151
5.1 Разработка простых представлений . . . 152
*5.1.1 Выбор кристально ясных имен* . . . 153
*5.1.2 Выбор простых в использовании типов данных и форматов* . . . 155
*5.1.3 Выбор готовых к использованию данных* . . . 157
5.2 Проектирование простых взаимодействий . . . 159
*5.2.1 Запрос простых входных данных* . . . 160
*5.2.2 Выявление всех возможных ошибок* . . . 162
*5.2.3 Возвращение информативного сообщения об ошибке* . . . 163
*5.2.4 Возвращение исчерпывающего сообщения об ошибке* . . . 168
*5.2.5 Возвращение информативного сообщения об успешном результате* . . . 170
5.3 Проектирование простых потоков . . . 172
*5.3.1 Построение простой цепочки целей* . . . 174
*5.3.2 Предотвращение ошибок* . . . 176
*5.3.3 Объединение целей* . . . 178
*5.3.4 Проектирование потоков без сохранения состояния* . . . 180

**6** *Проектирование предсказуемого API* ....183
6.1 Согласованность ....184
6.1.1 *Проектирование согласованных данных* ....185
6.1.2 *Проектирование согласованных целей* ....188
6.1.3 *Четыре уровня согласованности* ....189
6.1.4 *Копируя других: следование общепринятым практикам и соблюдение стандартов* ....190
6.1.5 *Согласованность – это сложно, и все нужно делать по-умному* ....194
6.2 Адаптируемость ....195
6.2.1 *Предоставление и принятие разных форматов* ....195
6.2.2 *Интернационализация и локализация* ....199
6.2.3 *Фильтрация, разбиение на страницы и сортировка* ....202
6.3 Быть видимым ....204
6.3.1 *Предоставление метаданных* ....205
6.3.2 *Создание гипермедиа-API* ....206
6.3.3 *Использование преимуществ протокола HTTP* ....210

**7** *Проектирование лаконичного и хорошо организованного API* ....213
7.1 Организация API ....213
7.1.1 *Организация данных* ....215
7.1.2 *Организация ответных сообщений* ....217
7.1.3 *Организация целей* ....219
7.2 Определение размера API ....225
7.2.1 *Выбор детализации данных* ....226
7.2.2 *Выбор детализации целей* ....228
7.2.3 *Выбор детализации API* ....229

**Часть III. Контекстное проектирование API** ....233

**8** *Проектирование безопасного API* ....235
8.1 Обзор безопасности API ....237
8.1.1 *Регистрация потребителя* ....237
8.1.2 *Получение учетных данных для использования API* ....238
8.1.3 *Выполнение API-вызова* ....240
8.1.4 *Проектирование API с точки зрения безопасности* ....241
8.2 Разделение API на части для облегчения управления доступом ....243
8.2.1 *Определение гибких, но точных групп* ....245
8.2.2 *Определение простых, но менее детализированных групп* ....247
8.2.3 *Выбор стратегии* ....250
8.2.4 *Определение групп с помощью формата описания API* ....251
8.3 Проектирование с учетом управления доступом ....254

*8.3.1 Какие данные необходимы для управления доступом* . . . . . . 254
*8.3.2 Адаптация дизайна при необходимости* . . . . . . . . . . . . . . . . . . 255
8.4 Обработка конфиденциальных данных и важных вещей . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 257
*8.4.1 Обработка конфиденциальных данных* . . . . . . . . . . . . . . . . . . 258
*8.4.2 Обработка конфиденциальных целей* . . . . . . . . . . . . . . . . . . . . 261
*8.4.3 Проектирование безопасных сообщений об ошибках* . . . . . . 264
*8.4.4 Выявление проблем, связанных с архитектурой и протоколом* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 266

**9** ***Изменение дизайна API*** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 269
9.1 Проектирование изменений API . . . . . . . . . . . . . . . . . . . . . . . 271
*9.1.1 Избегая критических изменений в выходных данных* . . . . . . 272
*9.1.2 Как избежать критических изменений во входных данных и параметрах* . . . . . . . . . . . . . . . . . . . . . . 277
*9.1.3 Как избежать критических изменений в сообщениях об успехе или ошибках* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 281
*9.1.4 Как избежать критических изменений в целях и потоках* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 284
*9.1.5 Предотвращение нарушений в системе безопасности и критических изменений* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 286
*9.1.6 Невидимый контракт интерфейса* . . . . . . . . . . . . . . . . . . . . . . 287
*9.1.7 Критическое изменение – не всегда проблема* . . . . . . . . . . . . 288
9.2 Управление версиями API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 289
*9.2.1 Управление версиями API и реализации* . . . . . . . . . . . . . . . . . 290
*9.2.2 Выбор представления управления версиями API с точки зрения потребителя* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292
*9.2.3 Выбор детализации* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295
*9.2.4 Влияние управления версиями API за пределами проектирования* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 301
9.3 Проектирование API с учетом расширяемости . . . . . . . . . . . 302
*9.3.1 Проектирование расширяемых данных* . . . . . . . . . . . . . . . . . . 302
*9.3.2 Проектирование расширяемых взаимодействий* . . . . . . . . . 306
*9.3.3 Проектирование расширяемых потоков* . . . . . . . . . . . . . . . . . 307
*9.3.4 Проектирование расширяемых API* . . . . . . . . . . . . . . . . . . . . . . 308

**10** ***Проектирование эффективного API для сети*** . . . . . . 311
10.1 Обзор проблем передачи данных по сети . . . . . . . . . . . . . . . 312
*10.1.1 Подготовка сцены* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 313
*10.1.2 Анализ проблем* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 315
10.2 Обеспечение эффективности передачи данных по сети на уровне протокола . . . . . . . . . . . . . . . . . . . . . . . . . . 320
*10.2.1 Активация сжатия и постоянных соединений* . . . . . . . . . . . 320
*10.2.2 Активация кеширования и условных запросов* . . . . . . . . . . 321
*10.2.3 Выбор политики кеширования* . . . . . . . . . . . . . . . . . . . . . . . . . 325

10.3 Обеспечение эффективности передачи данных по сети на уровне дизайна . . . . . 326
*10.3.1 Активация фильтрации . . . . . 327*
*10.3.2 Выбор соответствующих данных для представлений списка . . . . . 330*
*10.3.3 Агрегирование данных . . . . . 332*
*10.3.4 Предложение разных представлений . . . . . 334*
*10.3.5 Активация расширения . . . . . 336*
*10.3.6 Активация запросов . . . . . 338*
*10.3.7 Предоставление более релевантных данных и целей . . . . . 340*
*10.3.8 Создание разных слоев API . . . . . 343*

**11** ***Проектирование API в контексте*** . . . . . 345
11.1 Адаптация передачи данных к целям и характеру данных . . . . . 347
*11.1.1 Управление длительными процессами . . . . . 347*
*11.1.2 Уведомление потребителей о событиях . . . . . 349*
*11.1.3 Потоковая передача событий . . . . . 352*
*11.1.4 Обработка нескольких элементов . . . . . 357*
11.2 Соблюдение полного контекста . . . . . 365
*11.2.1 Знание существующих практик и ограничений потребителей . . . . . 365*
*11.2.2 Тщательно учитываем ограничения поставщика . . . . . 370*
11.3 Выбор стиля API в соответствии с контекстом . . . . . 373
*11.3.1 Сравнение API на базе ресурсов, данных и функций . . . . . 374*
*11.3.2 За границами API на базе HTTP . . . . . 379*

**12** ***Документирование API*** . . . . . 381
12.1 Создание справочной документации . . . . . 384
*12.1.1 Документирование моделей данных . . . . . 385*
*12.2.1 Документирование целей . . . . . 389*
*12.1.3 Документирование безопасности . . . . . 396*
*12.1.4 Обзор API . . . . . 398*
*12.1.5 Генерирование документации из кода реализации: плюсы и минусы . . . . . 399*
12.2 Создание руководства пользователя . . . . . 400
*12.2.1 Документирование вариантов использования . . . . . 401*
*12.2.2 Документирование безопасности . . . . . 404*
*12.2.3 Предоставление обзора общепринятого поведения и принципов . . . . . 404*
*12.2.4 Мышление вне статической документации . . . . . 404*
12.3 Предоставление адекватной информации разработчикам . . . . . 405
12.4 Документирование изменений API и устаревшие функции . . . . . 409

**13 *Развитие API*** .....413
13.1 Жизненный цикл API .....414
13.2 Создание руководства по проектированию API .....415
*13.2.1 Что включить в руководство по проектированию API* .....416
*13.2.2 Постоянное создание руководств* .....420
13.3 Проверка API.....422
*13.3.1 Оспаривание потребностей и их анализ* .....424
*13.3.2 Линтирование* .....427
*13.3.3 Проверка дизайна с точки зрения поставщика* .....430
*13.3.4 Проверка дизайна с точки зрения потребителя* .....432
*13.3.5 Проверка реализации* .....433
13.4 Общайтесь и делитесь информацией .....435

# *Предисловие*

На протяжении более десяти лет для большинства разработчиков проектирование API всегда подразумевало архитектуру REST. Такая ситуация сложилась в связи с регулярным выходом книг и блогов об API, которые продвигают мировоззрение RESTful – и это порождает некую зацикленность на данном направлении, если не сказать догматизм. Эта книга знаменует появление руководств по проектированию API следующего поколения, которые помогут нам преодолеть ограничения, тормозившие более десяти лет. Практический подход к разработке API по-прежнему основывается на REST, но автор очень старался сформировать представление о разработке API в практических ситуациях за вычетом имеющихся догм.

Арно Лоре знакомит нас с основами проектирования API, которые легко можно найти в других отраслевых изданиях, и раскрывает тему просто и доступно, рассматривая широкий круг вопросов в удобной для читателя форме. Я знаком с Арно уже несколько лет и считаю его одним из немногих высокопрофессиональных специалистов, которые не только знают, как создавать API технически, но и понимают проблемы, связанные с их доставкой потребителю, и знают, в каких случаях API могут оказывать положительное или отрицательное влияние на опыт разработчика. Арно фокусируется не только на проектировании API, но и на предоставлении правильно спроектированного API целевой аудитории продуманным образом.

Я лично наблюдал, как Арно принимал самое активное участие в дискуссиях, касающихся API, по всему миру, извлекая из них все самое полезное для себя. Достаточно просмотреть его страницу в Twitter или перейти по хештегу, посвященному какому-нибудь популярному мероприятию, связанному с API, чтобы понять, о чем я говорю. Арно использует уникальный подход, слушая доклады представителей из индустрии API и обрабатывая информацию, которой они делятся, и затем освеща-

ет в Twitter важные моменты беседы, представляя сведения об API в удобном для восприятия формате. Я рад, что Арно свел накопленные им знания воедино и изложил в книге, продолжая не только оттачивать собственные навыки, но и делиться с другими людьми своими знаниями и уникальным подходом к проектированию API. Арно принадлежит к редкой породе API-аналитиков, которые заботятся о развитии темы: впитывают знания об API, усваивают их и распространяют в доступной форме, так что их становится действительно легко применить в мире бизнеса.

После того как, начиная с 2012 года, проектирование API стало набирать обороты и стала очевидной доминирующая роль спецификации OpenAPI (ранее известной как Swagger), Арно был одним из немногих экспертов в области API, которые усердно старались раскрыть потенциал этой спецификации, а также разрабатывали инновационные инструменты и визуализации, связанные со стандартом спецификации API с открытым исходным кодом – чтобы понять не только спецификацию, но и то, как она может реализовывать, представлять и даже организовывать многие принципы проектирования API, необходимые для достижения успеха. Понадобится много работать, прежде чем вы поймете, что OpenAPI – не только документация; большинство разработчиков API так и не проходят этот путь до конца. Арно понимает, что OpenAPI – это к тому же и основа проектирования API для любой платформы, что помогает определить каждую остановку на протяжении всего жизненного цикла API. Эта книга – первое руководство по проектированию API из тех, что я видел, в котором OpenAPI и проектирование API объединены так вдумчиво и эффективно, что многие разработчики почувствуют несомненную пользу чтения.

Потратьте время, чтобы понять, чем Арно делится с вами. Это книга не предназначена для беглого пролистывания, и она уж точно не из серии «прочитал и забыл». Это справочник. Руководство по переходу проектирования ваших API на новый уровень. Оно дает вам набор концепций API, с которыми вы знакомы, и предоставляет вам чертежи для создания «Тысячелетнего сокола» или даже «Звезды Смерти» (если, конечно, пожелаете) из набора строительных блоков API.

Рекомендую вам прочитать эту книгу, а затем отложить ее на месяц. Затем приступайте к созданию API и переходу от проектирования к фактическому развертыванию и общедоступности – и поделитесь своими знаниями с разработчиками. А пока ждете отзывов, сядьте и снова возьмитесь за книгу. Вы начнете понимать глубину и ценность знаний, которыми Арно делится с вами. Возвращайтесь к чтению до тех пор, пока вы не научитесь в полной мере создавать API – не идеальные образцы,

а именно тот API, который вам нужен, чтобы достучаться до потребителей, на которых вы рассчитываете повлиять.

*— Кин Лейн, апологет API*

# От автора

На протяжении своей карьеры я большей частью соединял программные блоки, используя различные технологии программных интерфейсов – начиная с простых файлов и баз данных и заканчивая удаленными программными интерфейсами на базе RPC, Corba, Java RMI, веб-сервисов SOAP и веб-API. В эти годы мне посчастливилось работать над разнородными распределенными системами, смешивая очень старую технологию мейнфреймов с современными облачными системами и всем, что находится посередине между ними. Мне также посчастливилось работать с обеими сторонами программных интерфейсов в различных контекстах. Я работал над IVR (Interactive Voice Response), веб-сайтами и мобильными приложениями, созданными поверх огромных систем сервис-ориентированной архитектуры. Я создавал закрытые и общедоступные веб-сервисы и веб-API для веб-приложений и приложений на стороне сервера. Все эти годы я много жаловался на ужасные программные интерфейсы – и попадал в ловушки, создавая точно такие же.

Шли годы, и технология развивалась, переходя от RPC к веб-сервисам SOAP и веб-API. Объединять программное обеспечение становилось все проще и проще с технической точки зрения. Но независимо от используемой технологии, я понял, что программный интерфейс – это гораздо больше, чем просто связующая система или побочный продукт программного проекта. После посещения первых конференций по API в 2014 году на API Days в Париже я понял, что есть множество других людей, которые, как и я, сражаются с API. Вот почему в 2015 году я начал вести блог API Handyman, а также стал участвовать в конференциях, посвященных API. Я хотел поделиться своим опытом с другими и помочь им избежать попадания в те же ловушки, в которые попал я. Когда я писал о веб-API и обсуждал их, это не только позволяло мне помогать другим, но и давало мне возможность узнать о них еще больше.

После двух лет ведения блогов и выступлений на конференциях я пришел к решению написать книгу. Я хотел адресовать ее прежнему себе – тому, кто испытывал столько затруднений. К счастью, издательство Manning Publications искало человека, готового написать книгу о спецификации OpenAPI, формате описания API (об этом мы поговорим в главе 4). Я воспользовался шансом, предложив свою книгу Design of Everyday APIs, и ее приняли. На это название меня вдохновила книга «Дизайн привычных вещей» Дона Нормана (MIT Press, 1998) – вам обязательно стоит прочитать ее. Позже первоначальный заголовок заменили более простым, The Design of Web APIs. Должен признать, что он мне нравится больше; теперь у меня нет ощущения, что я покушаюсь на лавры Дона Нормана.

Поначалу в книге освещалось проектирование повседневных вещей + API + REST в сравнении с gRPC в сравнении с GraphQL. Усвоить все это было бы довольно трудно, но я хотел, чтобы изложенные в книге принципы можно было использовать для любого типа API. Месяц за месяцем содержание совершенствовалось, в результате чего получилась книга, которая сейчас называется The Design of Web APIs («Проектирование веб-API»). Я решил сосредоточиться на REST API и использовать их в качестве примера для изучения принципов проектирования веб- и удаленных API, что выходит за рамки простого проектирования API. Если бы такая книга попалась в прежние времена, я был бы очень рад ее изучить; надеюсь, вам тоже понравится!

# *Благодарности*

Два года. Мне потребовалось два года, чтобы написать книгу. Да, это немало, за-то в итоге получилась замечательная книга, которая, надеюсь, вам понравится. Я работал над ней не в одиночку. Мне многих нужно поблагодарить за непосредственную помощь в работе, но не меньше я благодарен и тем, кто эту работу сделал возможной. Прежде всего я благодарен своей жене Синзии и дочери Элизабетте. Большое вам спасибо за вашу поддержку и терпение, пока я проводил вечера и выходные за написанием книги. Я очень вас люблю.

Далее я хотел бы поблагодарить всех сотрудников издательства Manning Publications. Многие даже не представляют, сколько людей работает над книгой с самого начала ее создания! Каждый из этих людей проделал замечательную работу. Главным образом я бы хотел поблагодарить моего редактора Майка Стивенса, который поверил в этот проект. Особая благодарность двум редакторам – консультантам по аудитории, Кевину Харрелду и Дженнифер Стаут, и техническому редактору Майклу Ланду. Вы действительно очень помогли мне! Без вашего участия эта книга не стала бы такой, какой ее видит читатель. И отдельное merci beaucoup моему редактору текста из ESL Рэйчел Хэд, которая проделала поистине потрясающую работу, корректируя мой «французский английский». Спасибо производственному редактору Дейрдре Хайам, редактору Фрэнсис Буран, корректору Мелоди Долаб и техническому редактору Полу Гребенцу. Также я хотел бы поблагодарить своих рецензентов: Эндрю Гвоздзевич, Энди Кирша, Бриджера Хауэлла, Эдвина Квока, Энрико Маццареллу, Мохаммада Али Бацци, Нараянан Джаяратчаган, Питера Пола Селларса, Равиша Шарму, Санджива Кумара Джайсвала, Серхио Пачеко, Шона Хиксона, Шона Смита, Винсента Терона и Уильяма Руденмальма.

Особая благодарность Ивану Гончарову, переславшему мне 15 марта 2017 года электронное письмо от издательства Manning Publications. Оно искало человека, который написал бы книгу об API, в результате чего

и появилась на свет «Проектирование веб-API». Я рад, что по счастливой случайности мы встретились на REST Fest 2015.

Спасибо всем, кто нашел время, чтобы прочитать рукопись на разных этапах ее подготовки и предоставил неоценимую поддержку и отзывы. Отдельная благодарность Изабель Реуса и Мехди Меджауи за «полевое тестирование» содержания книги и за обратную связь. И спасибо всем тем, кто занимается проектированием API, – людям, с которыми я встречался и работал на протяжении многих лет. Я вложил в эту книгу все, чему научился у вас!

И еще один раз поблагодарю Майка Амундсена, Кина Лэйна и Мехди Меджауи, теперь уже за поддержку и помощь в то время, когда я начал вести блог API Handyman (в 2015 году). Без вас этой книги бы не было.

# Об этой книге

Эта книга была написана, для того чтобы помочь вам проектировать веб-API, которые не просто удовлетворяют выраженные потребности. Книга поможет вам проектировать выдающиеся веб-API, которые могут использоваться любым человеком в самых разных контекстах и которые также являются безопасными, долговечными, расширяемыми, эффективными и реализуемыми. В ней раскрываются все аспекты проектирования веб-API и дается полный обзор экосистемы веб-API и того, как проектировщики API могут внести в нее свой вклад.

### Кому адресована эта книга?

Данная книга, очевидно, предназначена для всех тех, кому необходимо проектировать веб-API. Это могут быть разработчики, работающие над серверной частью для мобильных приложений или веб-сайтов, или те, кому нужно соединять микросервисы воедино, или же это могут быть владельцы продуктов, работающие над API в качестве продукта и всем, что находится посередине. На самом деле эту книгу могут прочитать все те, кто работает над проектом, связанным с созданием API.

### Как устроена эта книга: дорожная карта

Книга состоит из трех частей, которые охватывают 13 глав.

В первой части представлены основные понятия и навыки, необходимые для проектирования API.

В главе 1 обсуждается, что такое API, чем важно его проектирование, и что из себя представляют элементы проектирования API.

В главе 2 объясняется, как точно определить назначение API – его реальные цели, – сосредоточив внимание на точке зрения пользователей API и программного обеспечения, использующего API, и избегая точки зрения организации и программного обеспечения, предоставляющего API.

Глава 3 знакомит вас с протоколом HTTP, REST API и архитектурным стилем REST. Она учит, как проектировать интерфейс веб-программирования (содержащий ресурсы, действия над ресурсами, данные, параметры и ответы) на основе определенных целей.

Глава 4 знакомит вас со спецификацией OpenAPI и демонстрирует, как описывать API структурированным и стандартным способом, используя такой формат описания API.

Вторая часть посвящена тому, как проектировать API типа «не заставляйте меня думать», которые будут просты для понимания и использования.

В главе 5 объясняется, как проектировать простые представления данных, сообщения об ошибках и успешных результатах, а также потоки вызовов API, которые сразу же будут понятны, и их легко будет использовать.

В главе 6 рассказывается, как создавать еще более простые для понимания и удобные в использовании API-интерфейсы, пользователи которых (люди или машины) смогут угадать, как работают API, делая их согласованными, адаптируемыми и обнаруживаемыми.

В главе 7 показано, как организовать и оценить все аспекты API, чтобы их было легко понять и использовать.

В третьей части показано, что проектировщики API должны учитывать контекст, окружающий API, и весь контекст, окружающий сам процесс проектирования API.

В главе 8 описывается безопасность API и то, как проектировать безопасные API.

В главе 9 рассказывается, как изменить API, не оказывая чрезмерного влияния на его пользователей, и как и когда использовать управление версиями. В ней также демонстрируется, как проектировать API, которые будет легко расширять с нуля.

Глава 10 посвящена тому, как создавать веб-API для эффективной работы в сети.

Глава 11 раскрывает весь контекст, который должны учитывать проектировщики при проектировании API. Она включает в себя адаптацию механизмов обмена данными (запрос/ответы, асинхронность, события и пакетную обработку), оценку и адаптацию к ограничениям потребителей или поставщиков и выбор адекватного стиля API (на базе ресурсов, функций или данных).

В главе 12 объясняется, как проектировщики API участвуют в создании различных типов документации API, используя преимущества такого формата описания API как спецификацию OpenAPI.

В главе 13 показано, как проектировщики могут участвовать в развитии множества API, принимая участие в жизненном цикле API. Особое внимание уделяется руководству по проектированию API и обзору.

Эту книгу следует читать от корки до корки, каждую главу по порядку. Каждая новая глава расширяет то, что было изучено в предыдущих. При этом, как только вы закончите главы 1, 2 и 3, вы можете перейти к любой главе, посвященной теме, которую вам необходимо срочно исследовать.

### О коде

В данной книге содержится множество примеров исходного кода как в пронумерованных списках, так и в обычном тексте. В обоих случаях исходный код форматируется `шрифтом фиксированной ширины` наподобие этого, чтобы отделить его от обычного текста. Иногда код также печатается **жирным шрифтом**, чтобы выделить код, который изменился по сравнению с предыдущим вариантом, например когда в существующую строку кода добавляется новая функция.

Во многих случаях оригинальный исходный код был переформатирован; мы добавили разрывы строк и переработали отступы, чтобы разместить доступное пространство страницы в книге. В редких случаях даже этого было недостаточно, и листинги содержат маркеры продолжения строки (➥). Кроме того, комментарии в исходном коде часто удаляются из листингов, когда описание кода приводится в тексте. Многие листинги сопровождаются кодовыми аннотациями для выделения важных понятий.

Исходный код примеров из этой книги доступен для скачивания с веб-сайта издателя по адресу https://www.manning.com/books/the-design-of-web-apis.

### Дискуссионный форум liveBook

Покупка книги *The Design of Web APIs* дает право свободного доступа к закрытому веб-форуму издательства Manning Publication, где можно высказать свои замечания о книге, задать технические вопросы и получить помощь от авторов и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке https://livebook.manning.com/book/the-design-of-everyday-apis/welcome/v-11/discussion. Вы также можете узнать больше о форумах издательства и правилах поведения на них по адресу https://livebook.manning.com/discussion.

Издательство Manning обязуется предоставить читателям площадку для содержательного диалога не только между читателями, но и между читателями и авторами. Но авторы не обязаны выделять определенное количество времени для участия, т. к. их вклад в работу форума является добровольным (и неоплачиваемым). Мы предлагаем читателям задавать авторам действительно непростые вопросы, чтобы их интерес не угасал! Форум и архив предыдущих обсуждений будут доступны на веб-сайте издательства, пока книга находится в печати.

### Другие интернет-ресурсы

Существует множество онлайн-ресурсов об API. Вот два моих любимых:

- *API Developer Weekly Newsletter* (https://apideveloperweekly.com/) – лучший способ узнать, что происходит в мире API и открыть для себя новые источники информации об API;
- сайт Web API Events (https://webapi.events/), он будет информировать вас о предстоящих конференциях, посвященных API.

### *Отзывы и пожелания*

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

### *Скачивание исходного кода примеров*

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

### *Список опечаток*

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

### *Нарушение авторских прав*

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

# Об авторе

*Арно Лоре* – архитектор программного обеспечения с 17-летним опытом из Франции. Большую часть этого периода он провел в финансовом секторе, работая над объединением систем различными способами, особенно с использованием веб-сервисов и API. Он ведет блог *API Handyman* и веб-сайт API Stylebook. Его приглашают в качестве лектора на конференции по API по всему миру. Арно увлекается -откой программного обеспечения, ориентированного на человека, и любит создавать и помогать создавать системы, обеспечивающие замечательный опыт для всех пользователей – от разработчиков и рабочих групп до конечных пользователей.

# *Об иллюстрации на обложке*

Рисунок на обложке книги носит заголовок «Девушка из Дрниша, Далмация, Хорватия». Эта иллюстрация взята из репродукции альбома традиционных хорватских костюмов середины XIX века Николы Арсеновича, опубликованного Этнографическим музеем в городе Сплите, Хорватия, в 2003 году. Иллюстрации были любезно предоставлены библиотекарем из этого музея, который расположен в римской части средневекового центра города: руинах дворца императора Диоклетиана, датируемых примерно 304 годом н. э. В книгу включены цветные иллюстрации жителей разных регионов Хорватии, сопровождаемые описаниями костюмов и повседневной жизни.

За последние 200 лет дресс-код и образ жизни изменились, а разнообразие, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных деревнях или городах, разделенных всего несколькими милями. Возможно, разнообразие культур преобразовалось в разнообразие личной жизни – и уж конечно, в более разнообразную и динамичную технологичную жизнь.

Издательство Manning отмечает изобретательность и инициативность компьютерных технологий обложками книг, на которых представлено разнообразие региональных культур два столетия назад. Эти культуры оживают в нашей памяти благодаря иллюстрациям из старых книг и коллекциям, подобным этой.

# Часть I

# Основы проектирования API

Каждое путешествие начинается с первого шага, и проектирование API не является исключением. Проектировщикам API необходимо обладать множеством навыков и нужно учитывать множество тем при создании API, но без прочной основы все передовые навыки и темы ничего не стоят. Это то, о чем вы узнаете в первой части.

Сначала мы рассмотрим ситуацию, объяснив, что такое API, почему его нужно проектировать и что на самом деле означает *обучение проектированию API*. Вы узнаете, что, будучи программными интерфейсами, API – это больше, чем просто «связующее звено», и что для проектирования любого типа API необходимо изучить фундаментальные принципы.

Еще до того, как задуматься о программировании, вы увидите, что API нужно рассматривать с точки зрения его пользователей. Предполагается, что API позволяет вашим пользователям легко достигать своих целей, а не системам, представляющим API. Только после того, как эти цели станут известны и точно описаны, можно спроектировать фактический интерфейс программирования, такой как REST API. И, как и любое программирование, описание интерфейса программирования должно выполняться с помощью адаптированного инструмента, такого как спецификация OpenAPI для REST API.

# Что такое проектирование API

**В этой главе вы узнаете:**

- что такое API;
- почему важно проектирование API;
- что означает проектирование API.

Веб-API (программные интерфейсы приложения) являются неотъемлемой частью нашего взаимосвязанного мира. Программное обеспечение использует эти интерфейсы для обмена данными – от приложений на смартфонах до глубоко скрытых серверов баз данных, API-интерфейсы присутствуют абсолютно везде. Считаются ли они простыми техническими интерфейсами или продуктами сами по себе, целые системы, независимо от размера и назначения, зависят от них. То же самое делают целые компании и организации, начиная с технологических стартапов и интернет-гигантов, заканчивая нетехническими малыми и средними предприятиями, крупными корпорациями и государственными структурами.

Если API-интерфейсы являются неотъемлемой частью нашего взаимосвязанного мира, их проектирование – это его основа. При создании и развитии системы на базе API, независимо от того, является ли она видимой для кого-либо или глубоко скрытой, создает ли она один или несколько API-интерфейсов, проектирование всегда должно быть основной задачей. Успех или неудача такой системы напрямую зависит от качества проектирования всех ее API. Но что на самом деле означает проектирование API? И что нужно изучать, чтобы проектировать

их? Чтобы ответить на эти вопросы, нужно рассмотреть, что такое API и для кого они предназначены, а также понять, что проектирование API – это больше, чем просто проектирование программного интерфейса для приложений.

## 1.1. Что такое API?

Миллиарды людей владеют смартфонами и используют их для обмена фотографиями в социальных сетях. Без API это было бы невозможно. Обмен фотографиями с помощью мобильного приложения для социальных сетей предполагает использование различных типов API, как показано на рис. 1.1.

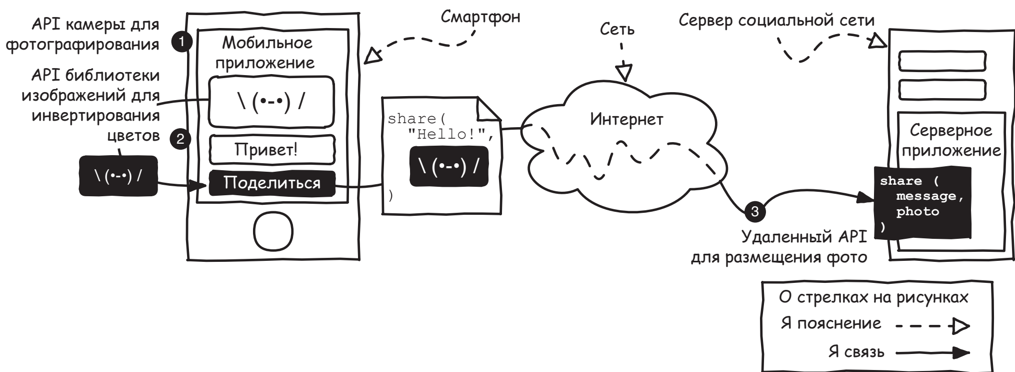



**Рис. 1.1.** Три разных типа API

Во-первых, чтобы сделать фото, мобильное приложение социальной сети использует камеру смартфона с помощью API. Затем через API оно может использовать некую библиотеку изображений, встроенную в приложение, для инвертирования цветов фотографии. И в итоге оно делится измененной фотографией, отправляя ее на серверное приложение, размещенное на сервере социальной сети, используя удаленный API, доступ к которому можно получить через сеть, обычно через интернет. Таким образом, в этом сценарии задействованы три различных типа API: аппаратный API, библиотека и удаленный API, соответственно. Эта книга посвящена последнему типу.

API-интерфейсы, независимо от своего типа, упрощают создание программного обеспечения, но удаленные API, особенно веб-API, изменили способ создания программного обеспечения. В настоящее время любой может легко создать что-либо, собрав удаленные части программного обеспечения. Но, прежде чем говорить о безграничных возможностях, предоставляемых такими API, давайте разберемся, что на самом деле подразумевается под термином *API* в этой книге.

### 1.1.1. API – это веб-интерфейс для программного обеспечения

В этой книге API – это удаленный API, а точнее, веб-API – веб-интерфейс для программного обеспечения. API, независимо от типа, прежде всего

является интерфейсом: точкой, где две системы, субъекта, организации и т. д. встречаются и взаимодействуют. Поначалу концепция API может быть непростой для понимания, но рис. 1.2 делает ее более осязаемой, сравнивая ее с пользовательским интерфейсом приложения.

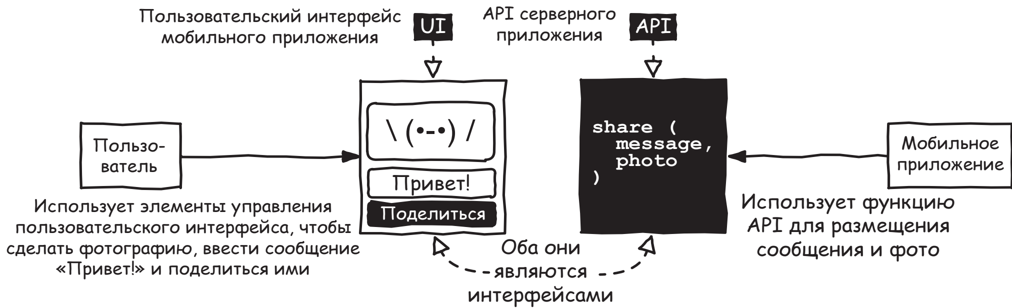



**Рис. 1.2.** Сравнение пользовательского интерфейса приложения с программным интерфейсом приложения (API)

Будучи пользователем мобильного приложения, вы взаимодействуете с ним, касаясь экрана вашего смартфона, на котором отображается пользовательский интерфейс приложения. Пользовательский интерфейс мобильного приложения может предоставлять такие элементы, как кнопки, текстовые поля или метки на экране. Эти элементы позволяют пользователям взаимодействовать с приложением для просмотра или предоставления информации, а также для запуска таких действий, как обмен сообщениями и фотографиями.

Так же как мы (люди) используем пользовательский интерфейс приложения для взаимодействия с ним, это приложение может использовать еще одно приложение с помощью своего программного интерфейса. Принимая во внимание, что пользовательский интерфейс предоставляет поля ввода данных, метки и кнопки, чтобы обеспечить обратную связь, которая может развиваться по мере их использования, API предоставляет функции, которые могут нуждаться во входных данных или которые могут возвращать выходные данные в качестве ответа. Эти функции позволяют другим приложениям взаимодействовать с приложением, предоставляющим API, для извлечения или отправки информации или для запуска действий.

Строго говоря, API – это *только* интерфейс, предоставляемый неким программным обеспечением. Это абстракция базовой *реализации* (базовый код – это то, что на самом деле происходит внутри программного продукта при использовании API). Но обратите внимание, что термин *API* часто используется для обозначения всего программного продукта, включая API и его реализацию.

Таким образом, API – это интерфейсы для программного обеспечения, но API, о которых мы говорим в этой книге, – это больше, чем просто API: это веб-API, как показано на рис. 1.3.

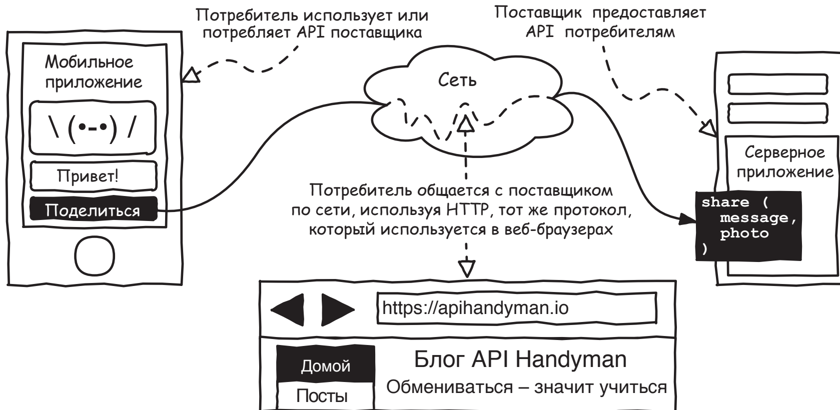



**Рис. 1.3.** Веб-API – это удаленные API, которые можно использовать с протоколом HTTP

Мобильное приложение, работающее в смартфоне, использует API, предоставляемый серверным приложением (часто именуемым бэкенд-приложением или просто *серверной частью*), которое размещено на удаленном сервере. Поэтому мобильное приложение называется *потребителем*, а серверная часть называется *поставщиком*. Данные термины также применимы к компаниям и людям, создающим приложения, или использующим или предоставляющим API. Здесь это означает, что разработчики мобильного приложения являются потребителями, а разработчики бэкенд-приложения – поставщиками.

Для обмена данными со своим бэкенд-приложением мобильное приложение обычно использует известную сеть: интернет. Здесь интересен не сам интернет – такой обмен данными также может осуществляться через локальную сеть, – а то, *как* эти два приложения обмениваются данными по сети. Когда мобильное приложение отправляет фотографию и сообщение бэкенд-приложению, оно использует Протокол передачи гипертекста (HTTP). Если вы открываете веб-браузер на компьютере или смартфоне, то используете HTTP (косвенно). Это протокол, который применяется любым веб-сайтом. Когда вы вводите адрес веб-сайта, например http://apihandyman.io или его защищенную версию, https://apihandyman.io, в адресной строке и нажимаете **Enter** или щелкаете по ссылке в браузере, браузер использует HTTP для связи с удаленным сервером, на котором размещен веб-сайт, чтобы показать вам содержимое сайта. Удаленные API или, по крайней мере, те, о которых мы говорим в этой книге, используют этот протокол так же, как веб-сайты; вот почему они называются *веб-API*.

Итак, в этой книге API – это веб-API. Это веб-интерфейсы для программного обеспечения. Но чем они так интересны?

### 1.1.2. API превращают программное обеспечение в детали конструктора LEGO®

Тысячи, даже миллионы мобильных приложений и их бэкендов были созданы благодаря веб-API, но это еще не все. Веб-API развивают твор-

ческий потенциал и инновации, превращая программное обеспечение в многократно используемые блоки, которые можно легко собрать. Давайте вернемся к нашему примеру и посмотрим, что может произойти, если разместить фото в социальной сети.

Когда бэкенд социальной сети получает фотографию и сообщение, он может сохранить фотографию в файловой системе сервера, а сообщение и идентификатор фотографии (для последующего извлечения фактического файла) – в базе данных. Он также может обработать фотографию, используя некий самодельный алгоритм распознавания лиц, чтобы определить, есть ли ней ваши друзья, перед тем как сохранить фотографию. Это одна из возможностей – одно приложение, обрабатывающее все для другого отдельного приложения. Давайте рассмотрим что-нибудь другое, как показано на рис. 1.4.

Внутренний API может использоваться как мобильным приложением социальной сети, так и веб-сайтом, и его реализация может быть совершенно иной. Когда серверная часть получает фотографию и сообщение для совместного использования (какое бы приложение его ни отправляло), она может делегировать хранение фотографии в качестве сервисной компании через свой API. Он также может делегировать хранение идентификатора сообщения и фотографии внутреннему программному модулю хроники через его API. Как обрабатывать распознавание лиц? Что же, это можно было бы делегировать какому-нибудь специалисту по распознаванию лиц, предлагающему свои услуги через... как вы уже догадались... API.

**Рис. 1.4. Система, состоящая из открытых и закрытых программных деталей LEGO®, подключенных через API**

Обратите внимание, что на рис. 1.4 каждый API предоставляет только одну функцию. Это делает рисунок максимально простым: один API мо-

жет предоставлять множество функций. Серверная часть может, например, предоставлять такие функции, как **Добавить друга**, **Перечислить друзей** или **Получить хронику**.

Это похоже на программную версию системы деталей конструктора LEGO; это те пластиковые блоки, которые можно собирать, чтобы создавать новые вещи. (Аристотель, вероятно, играл с некоторыми из них, когда понял, что «целое больше, чем сумма его частей».) Возможности тут бесконечны – единственным ограничением является ваше воображение.

Когда я был ребенком, я целыми часами играл в LEGO, собирая здания, машинки, самолеты, космические корабли или все что угодно. Когда мне надоедало одно из моих творений, я мог полностью уничтожить его и начать что-то новое с нуля или мог переделать его, заменив некоторые детали. Я мог бы даже собрать существующие структуры вместе, например чтобы создать массивный космический корабль. То же самое происходит и в мире API: вы можете разложить огромные системы программных блоков, которые можно легко собрать и даже заменить благодаря API, но есть небольшие отличия.

Каждый программный блок может одновременно использоваться и многими другими. В нашем примере внутренний API может использоваться мобильным приложением и веб-сайтом. Обычно API предназначен для использования не одним потребителем, а множеством. Таким образом, вам не нужно все время приступать к повторной разработке.

Каждый программный блок может работать где угодно самостоятельно, если он подключен к сети, чтобы быть доступным через API. Это хороший способ управления производительностью и масштабируемостью; программный блок, такой как блок распознавания лиц на рис. 1.4, вероятно, потребует гораздо больше ресурсов обработки по сравнению с хроникой социальной сети. Если первым управляет компания Social Network, его можно установить на другом, выделенном и более мощном сервере, тогда как хроника Social Network работает на сервере поменьше. А доступность через простое сетевое соединение позволяет любому API, предоставленному кем-либо, использоваться кем-либо.

В мире API существует два типа блоков, представляющих два типа API: *открытые API* и *закрытые API*. Программные блоки распознавания лиц и хранения фотографий создаются и даже управляются не компанией Social Network, а третьими лицами. Предоставляемые ими API являются открытыми.

Открытые API-интерфейсы предлагаются *в качестве сервиса* или *продукта* другими; вы не создаете их, не устанавливаете и не запускаете – вы только используете их. Открытые API предоставляются всем, кто в них нуждается и готов принять условия стороннего поставщика. В зависимости от бизнес-модели провайдеров API такие API могут быть бесплатными или платными, как и любой другой программный продукт. Такие открытые API-интерфейсы раскрывают творческий потенциал и инновации, а также могут значительно ускорить создание всего того, о чем вы можете мечтать. Перефразируя Стива Джобса, можно сказать,

что для этого есть API. Зачем терять время, пытаясь изобрести колесо, которое все равно будет недостаточно круглым? В нашем примере компания Social Network решила сосредоточиться на своем основном опыте, соединяя людей, и делегировала распознавание лиц третьей стороне.

Но открытые API – это только вершина айсберга. Серверная часть и временная шкала были созданы компанией Social Network для собственного использования. API хроники используются только приложениями, созданными этой компанией, как показано на рис. 1.4. То же самое касается мобильного бэкенда, который используется мобильным приложением Social Network. Эти API являются *закрытыми*, и их – миллиарды. Закрытый API – это API, который вы создаете для себя: его используют только приложения, созданные вами или сотрудниками вашей команды, отдела или компании. В данном случае вы являетесь поставщиком и потребителем собственного API.

**ПРИМЕЧАНИЕ.** Вопрос открытости/закрытости не в том, *как* API предоставляется, а *кому*. Даже будучи размещенным в интернете, API мобильного бэкенда по-прежнему является закрытым.

Между *настоящими* закрытыми и открытыми API могут быть различные виды взаимодействия. Например, вы можете установить коммерческое готовое программное обеспечение, такое как система управления контентом (CMS) или система управления взаимоотношениями с клиентами (CRM), на собственные серверы (такая установка часто называется *on-premise*), и эти приложения могут (и даже должны!) предоставлять API. Эти API являются закрытыми, но вы не создаете их сами. Тем не менее вы можете использовать их по своему усмотрению, в особенности для подключения большего количества блоков к своим блокам. Возьмем другой пример. Вы можете предоставить некоторые из ваших API-интерфейсов клиентам или выбранным партнерам. Такие *почти открытые* API часто называют *партнерскими*.

Но независимо от ситуации API в основном превращают программное обеспечение в программные блоки многократного использования, которые вы или другие пользователи могут собрать, чтобы создать модульные системы, способные выполнять абсолютно все. Вот почему API так интересны. Но почему их разработка так важно?

## 1.2. Чем важно проектирование API

Даже если это полезно, API кажется лишь техническим интерфейсом для программного обеспечения. Так в чем состоит важность проектирования такого интерфейса?

API-интерфейсы используются программным обеспечением, это правда. Но кто создает программное обеспечение, которое их использует? Разработчики. Люди. Эти люди ожидают, что эти программные интерфейсы будут полезными и простыми, как и любой другой (хорошо спроектированный) интерфейс. Подумайте о своей реакции, когда вы сталкиваетесь с плохо спроектированным веб-сайтом или пользователь-

ским интерфейсом мобильного приложения. Что вы чувствуете, когда сталкиваетесь с плохо продуманными повседневными вещами, такими как пульт дистанционного управления или даже дверь? Это может вызвать у вас раздражение, вероятно, вы даже рассердитесь или разразитесь тирадой и вряд ли захотите когда-либо пользоваться ими. А в некоторых случаях плохо спроектированный интерфейс может быть даже опасным. Вот почему проектирование любого интерфейса имеет значение, и API не являются исключением.

### *1.2.1. Открытый или закрытый API – это интерфейс для других разработчиков*

В разделе 1.1.1 вы узнали, что потребителем API может быть либо программное обеспечение, использующее API, либо компания, либо отдельные лица, разрабатывающие это программное обеспечение. Все эти потребители важны, но первый, кого нужно принимать во внимание, – это разработчик.

Как вы уже видели, в этом примере с социальными сетями участвуют разные API. Существует модуль хроники, который управляет хранением данных и предоставляет закрытый API. И есть открытые (предоставляемые другими компаниями) API распознавания лиц и хранения фотографий. Бэкенд, который вызывает эти три API, не появляется сам по себе; он разработан компанией Social Network.

Например, чтобы использовать API распознавания лиц, разработчики пишут код в программном обеспечении Social Network, чтобы отправлять фотографии для распознавания лиц и обрабатывать результат обработки фотографий, как при использовании программной библиотеки. Эти разработчики не являются теми, кто создал API распознавания лиц, и они, вероятно, не знают друг друга, потому что они из разных компаний. Мы также можем представить, что мобильное приложение, веб-сайт, серверная часть и модуль хранения данных разрабатываются разными группами внутри компании. В зависимости от организации компании эти команды могут хорошо знать друг друга или не знать вообще. И даже если каждый разработчик в компании знает все маленькие секреты каждого API, который она разработала, неизбежно появятся новые разработчики.

Таким образом, будь то открытый или закрытый API независимо от того, по какой причине он создан и какой бы ни была организация компании, рано или поздно он будет использоваться другими разработчиками – людьми, которые не участвовали в создании программного обеспечения, предоставляющего API. Вот почему нужно сделать все, для того чтобы при написании кода этим новичкам было легко использовать API. Разработчики ожидают, что API будут полезны и просты, как и любой интерфейс, с которым им приходится взаимодействовать. Вот почему так важно проектирование API.

**Опыт разработчика**

API (DX) – это опыт, который получают разработчики при использовании API. Он охватывает множество различных тем, таких как регистрация (для использования API), документирование (чтобы узнать, что делает API и как его использовать) или поддержка (чтобы получить помощь при возникновении проблем). Но все усилия ничего не стоят, если не решен самый важный вопрос – проектирование API.

### *1.2.2 API создается, для того чтобы скрыть реализацию*

Проектирование API имеет значение, потому что, когда люди используют API, они хотят использовать его, не беспокоясь о мелочах, которые не имеют к ним никакого отношения. А для этого разработка должна скрывать детали реализации (что на самом деле происходит). Позвольте мне использовать реальную аналогию в качестве объяснения.

Скажем, вы решили пойти в ресторан. Как насчет французского, например? Когда вы идете в ресторан, то становитесь клиентом. Как клиент ресторана, вы читаете его меню, чтобы узнать, какие блюда можно заказать. Вы решаете попробовать миногу по-бордосски (знаменитое французское рыбное блюдо из области Гасконь). Чтобы заказать выбранную еду, вы говорите с (обычно очень приятным и дружелюбным) человеком, которого называют официантом или официанткой. Спустя некоторое время официант возвращается и приносит заказанное вами блюдо – миногу по-бордосски, – которое приготовили на кухне. Пока вы едите свой вкусный обед, можно задать вам два вопроса?

Первое: вы знаете, как приготовить миногу по-бордосски? Наверное, нет, и, возможно, по этой причине вы и идете в ресторан. И даже зная рецепт приготовления, вы, вероятно, не захотите этого делать, потому что это сложно и для этого требуются труднодоступные ингредиенты. Вы отправляетесь в ресторан за блюдом, которое не умеете или не хотите готовить.

Второе: знаете ли вы, что произошло между моментом, когда официант принял ваш заказ и когда принес его вам? Вы можете догадаться, что официант был на кухне, чтобы отдать заказ повару, который работает один. Этот повар очень старается и уведомляет официанта, когда блюдо будет готово, звоня в маленький колокольчик и крича: «Заказ для столика № 2 готов!» Но сценарий может немного отличаться.

Официант может использовать смартфон, чтобы принять ваш заказ, который мгновенно отображается на сенсорном экране на кухне. А там не одинокий повар, а целая бригада. Как только блюдо готово, один из членов бригады помечает ваш заказ как готовый на сенсорном экране, а официант получает уведомление на свой смартфон. Независимо от количества поваров, вы не знаете рецепт и ингредиенты, используемые для приготовления еды. Независимо от сценария, еда, которую вы заказали, поговорив с официантом, была приготовлена на кухне, и официант принес ее вам. В ресторане вы говорите только с официантом (или офи-

цианткой), и вам не нужно знать, что происходит на кухне. Какое отношение все это имеет к API? Самое прямое, как показано на рис. 1.5.

С точки зрения группы разработчиков мобильных приложений для социальных сетей, *размещение фото* с использованием бэкенд-API абсолютно одинаково, только бэкенд уже реализован. Разработчикам не нужно знать рецепт и его ингредиенты; они только предоставляют фото и сообщение. Им все равно, будет ли фотография проходить через распознавание изображений до или после добавления в хронику. Они также не заботятся о том, является ли серверная часть единственным приложением, написанным на языке Go или Java, которое обрабатывает все или использует другие API, написанные на NodeJS, Python или любом другом языке. Когда разработчик создает *потребительское приложение* (клиент), которое использует *приложение провайдера* (ресторан) через свой API (официант или официантка), чтобы сделать что-то (например, заказать еду), разработчик и приложение знают только об API; им не нужно знать, как сделать что-то самостоятельно или как программное обеспечение провайдера (кухня) будет это делать. Но скрыть реализацию недостаточно. Это важно, но это не то, что на самом деле нужно тем, кто использует API.

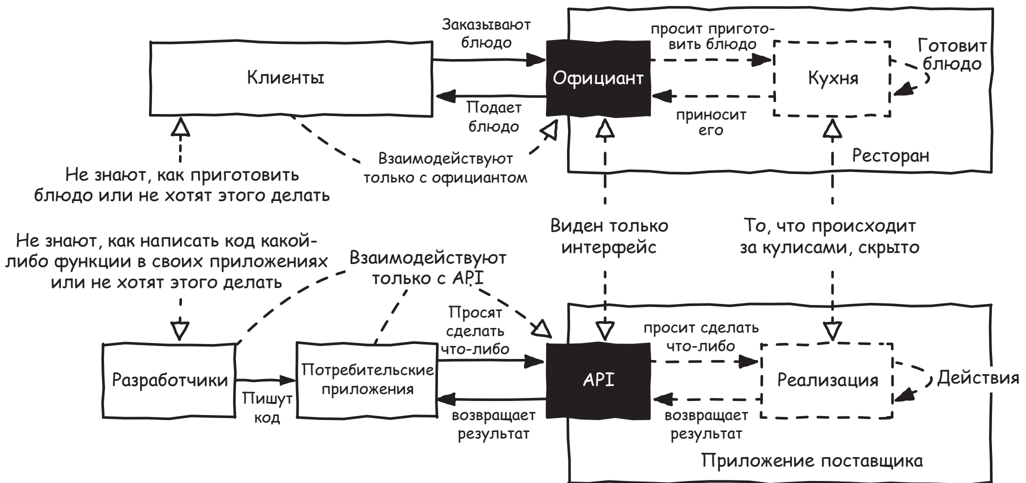



Рис. 1.5. Параллели между обедом в ресторане и использованием API

### 1.2.3. Страшные последствия плохо спроектированных API

Что вы делаете, когда впервые используете какую-либо повседневную вещь? Вы внимательно смотрите на ее интерфейс, чтобы определить ее назначение и то, как ее использовать, основываясь на том, что вы видите, и на своем прошлом опыте. И здесь важен дизайн.

Давайте рассмотрим гипотетический пример: устройство под названием UDRC 1138. Что это может быть за устройство? Какова его цель? Ну, его название не очень помогает. Возможно, его интерфейс может дать нам больше подсказок. Посмотрите на рис. 1.6.

**Рис. 1.6.** Загадочный интерфейс устройства UDRC 1138

Справа есть шесть немаркированных треугольных и прямоугольных кнопок. Может, это что-то вроде запуска и остановки? Ну, как кнопки медиаплеера. Четыре другие кнопки имеют незнакомую форму без малейшего намека на их назначение. На ЖК-дисплее отображаются номера без меток с такими единицами измерения, как ft, NM, rad и km/h. Можно предположить, что *ft* – это расстояние в футах, а *km/h* – скорость в километрах в час, но что могут означать *rad* и *NM*? Кроме того, в нижней части ЖК-экрана также появляется тревожное сообщение, предупреждающее о том, что мы можем вводить значения вне допустимого диапазона без контроля безопасности.

Этот интерфейс, безусловно, трудно расшифровать, и он не очень интуитивно понятен. Давайте посмотрим на документацию, изображенную на рис. 1.7, чтобы увидеть, что в ней говорится об этом устройстве.

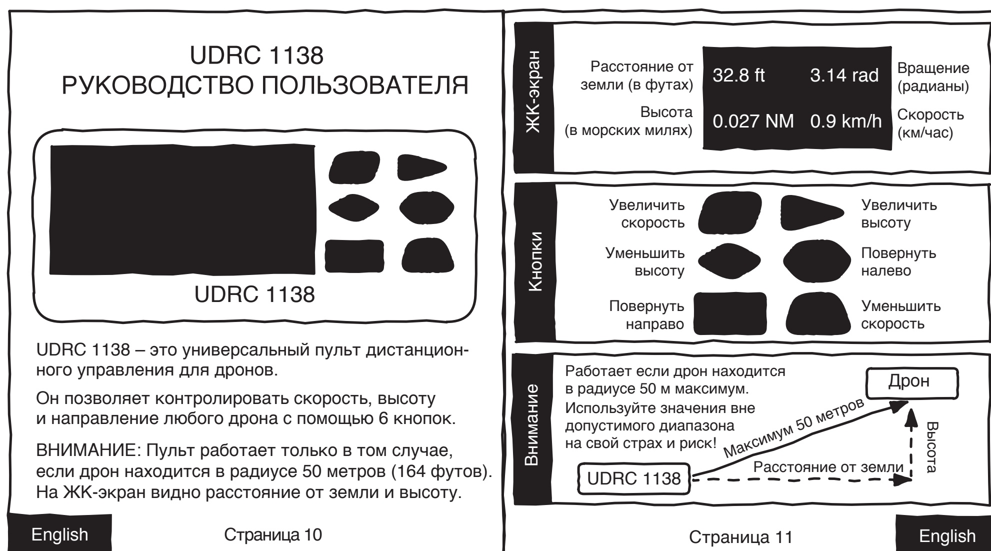



**Рис. 1.7.** Документация по UDRC 1138

Согласно описанию, это устройство представляет собой универсальный пульт дистанционного управления дроном – отсюда и название UDRC 1138, я полагаю, – который можно использовать для управления любым дроном. Ну что же, звучит интересно. На странице справа приве-

дено несколько пояснений по ЖК-экрану, кнопкам и сообщение с предупреждением.

Описание ЖК-экрана довольно загадочное. Расстояние от земли указано в футах, высота – в морских милях, ориентация дронов обеспечивается радианами, а скорость указана в километрах в час. Я не знаком с авиационными единицами измерения, но чувствую, что с выбранными единицами что-то не так. Они кажутся противоречивыми. Смесь футов и метров? И во всех фильмах о самолетах, которые я видел, футы используются для измерения высоты, а морские мили – для измерения расстояния, а не наоборот. Этот интерфейс определенно не интуитивно понятен.

Глядя на описания кнопки, видно, что для увеличения высоты используется треугольная кнопка в правом верхнем углу, а для ее уменьшения – ромбовидная кнопка во втором ряду. Связь неочевидна – почему эти элементы управления не идут друг за другом? Другие элементы управления также используют формы, которые абсолютно ничего не значат и кажутся расположенными случайным образом. Это безумие! Как использовать это устройство? Почему бы не взять старые добрые джойстики или геймпады вместо кнопок?

И последнее, но не менее важное объяснение предупреждающего сообщения. Похоже, что беспилотник может работать только в диапазоне 50 м – диапазон рассчитывается на основе расстояния до земли и высоты, обеспечиваемой ЖК-экраном. Подождите, что? Мы должны рассчитать расстояние между пультом дистанционного управления и дроном, используя теорему Пифагора?! Это полная чушь – устройство должно делать это за нас.

Вы бы купили или хотя бы попробовали в деле такой гаджет? Вероятно, нет. Но что, если у вас нет другого выбора, кроме как использовать это ужасное устройство? Что же, желаю удачи; вам понадобится что-то сделать с таким плохо спроектированным интерфейсом!

Вы, вероятно, думаете, что такой катастрофический интерфейс на самом деле не может существовать в реальной жизни. Конечно, проектировщики не могли создать такой ужасный прибор. И даже если бы они это сделали, конечно, отдел обеспечения качества никогда бы не пустил его в производство! Но давайте посмотрим правде в глаза. Плохо спроектированные устройства – вещи с плохо спроектированными интерфейсами для повседневного использования – все время *поступают* в производство.

Подумайте, сколько раз вы были озадачены или ворчали при работе с устройством, веб-сайтом или приложением, потому что в его дизайне были дефекты. Сколько раз вы решали не покупать или не использовать что-то из-за его дизайна? Сколько раз вы не могли применить что-то, или применить это правильно, или как вы хотели, потому что его интерфейс был непостижим?

Плохо спроектированный продукт может быть использован не по назначению, недоиспользован или не использован вообще. Это может даже быть опасно для его покупателей и для компании, которая созда-

ла его, и репутация такой компании может быть под угрозой. И если это физическое устройство, после того как оно будет запущено в производство, исправлять его будет слишком поздно.

Страшные недостатки дизайна не ограничиваются интерфейсами повседневных вещей. К сожалению, API также могут страдать от этого. Напомню, что API – это интерфейс, который разработчики должны использовать в своем программном обеспечении. Проектирование имеет значение независимо от типа интерфейса, и API-интерфейсы не являются исключением. Плохо спроектированные API-интерфейсы могут стать таким же разочарованием, что и устройства, подобные UDRC 1138; они могут быть очень трудным для понимания и использования, а это может иметь ужасные последствия.

Как люди выбирают открытый API, API в качестве продукта? Как и в повседневной жизни, они смотрят на его интерфейс и документацию. Они заходят на портал разработчиков API, читают документацию и анализируют API, чтобы понять, что он позволяет делать и как его использовать. Они оценивают, позволит ли он им эффективно и просто достичь своей цели. Даже самая совершенная документация не сможет скрыть недостатки проектирования, которые делают API трудным или даже опасным для использования. И если такие недостатки будут обнаружены, возможные пользователи (эти потенциальные клиенты) не станут выбирать этот API. Нет клиентов – нет дохода. Это может привести к банкротству компании.

Что, если кто-то решит воспользоваться некорректным API в любом случае? Иногда пользователи могут просто не обнаружить недостатки с первого взгляда. А иногда у них просто может не быть альтернативы. Такое может произойти, например, внутри какой-то компании. Чаще всего у людей нет другого выбора, кроме как использовать ужасные закрытые API или API, предоставляемые коммерческими готовыми приложениями.

Независимо от контекста недостатки проектирования увеличивают время, усилия и траты, необходимые для создания программного обеспечения с использованием API. API может быть использован неправильно или недоиспользован. Клиентам может потребоваться обширная поддержка со стороны поставщика API, что увеличивает затраты на стороне поставщика. Это потенциально серьезные последствия как для закрытых, так и для открытых API. Более того, благодаря открытым API-интерфейсам пользователи могут открыто пожаловаться или просто прекратить их применение, что приведет к уменьшению количества клиентов и снижению доходов для поставщиков API.

Ошибочное проектирование API также может привести к уязвимостям безопасности в API, таким как непреднамеренное раскрытие конфиденциальных данных, пренебрежение правами доступа и привилегиями группы или слишком большое доверие к потребителям. Что, если некоторые решат воспользоваться такими уязвимостями? Последствия для потребителей и поставщиков API могут быть катастрофическими.

В мире API плохое проектирование часто можно исправить, после того как API будет запущен в производство. Но издержки неизбежны:

поставщику понадобятся время и деньги, чтобы исправить ситуацию, и это может также серьезно беспокоить потребителей API.

Это лишь несколько примеров вредного воздействия. Плохо спроектированные API обречены на провал. Что можно сделать, чтобы избежать подобной участи? Все просто: *научиться правильно проектировать API*.

## 1.3. Элементы проектирования API

Научиться проектировать API – это гораздо больше, чем просто научиться проектировать программные интерфейсы. Обучение проектированию API требует изучения принципов, и не только технологий, но и знания всех аспектов проектирования. Для проектирования API важно не просто сосредоточиться на самих интерфейсах, но также знать весь контекст, окружающий их, и проявлять эмпатию ко всем пользователям и программному обеспечению. Проектирование API без принципов, полностью вне контекста и без учета обеих сторон интерфейса – как потребителя, так и поставщика, – лучший способ гарантировать полный провал.

### 1.3.1 Изучение принципов, выходящих за рамки проектирования программного интерфейса

Когда (хорошие) проектировщики помещают кнопку в определенное место, выбирают конкретную форму или решают добавить красный светодиод на объект, на то есть причина. Знание этих причин помогает проектировщикам создавать хорошие интерфейсы для повседневных вещей, которые позволят людям достичь своей цели как можно проще – будь то двери, стиральные машины, мобильные приложения или что-либо еще. То же самое происходит и в случае с API.

Цель API состоит в том, чтобы позволить людям достичь своих целей настолько просто, насколько это возможно, независимо от части, касающейся *программирования*. В программном обеспечении мода приходит и уходит. Было и будет много разных способов раскрытия данных и возможностей с помощью программного обеспечения. Было и будет много разных методов обеспечения обмена данными по сети. Возможно, вы слышали о RPC, SOAP, REST, gRPC или GraphQL. Вы можете создавать API-интерфейсы с помощью всех этих технологий: некоторые из них – это архитектурные стили, другие – протоколы или языки запросов. Чтобы было проще, будем называть их *стилями API*.

Каждый стиль API может сопровождаться некоторыми (более или менее) общепринятыми практиками, которым вы можете следовать, но они не защитят вас от ошибок. Не зная основополагающих принципов, вы можете растеряться при выборе так называемой общепринятой практики; вы можете изо всех сил пытаться найти решения, сталкиваясь с необычными случаями использования или контекстами, которые подобного рода практики не охватывают. И если вы переключитесь на новый стиль API, вам придется все изучать заново.

Знание основополагающих принципов проектирования API дает вам прочную основу для проектирования API любого стиля и решения лю-

бых задач проектирования. Но знание таких принципов – лишь один из аспектов проектирования.

### 1.3.2 Изучение всех аспектов проектирования API

Проектирование интерфейса – это гораздо больше, чем просто размещение кнопок на поверхности объекта. Создание такого объекта, как пульт дистанционного управления дроном, требует, чтобы проектировщики знали, какова его цель и чего хотят добиться люди, использующие его. Предполагается, что такое устройство должно контролировать скорость, высоту и направление летящего объекта. Это то, что нужно пользователям, и им все равно, будет ли это сделано с использованием радиоволн или любой другой технологии.

Все эти действия должны быть представлены в пользовательском интерфейсе с помощью кнопок, джойстиков, ползунков или других элементов управления. Назначение этих элементов управления и их представлений должно иметь смысл, чтобы у пользователей не возникало проблем при их применении, и, что самое важное, они должны быть полностью безопасными. Интерфейс UDRC 1138 является прекрасным примером абсолютно непригодного и небезопасного интерфейса с его ЖК-дисплеем, который сбивает с толку, или с кнопками и отсутствием управления безопасностью.

Конструкция такого пульта дистанционного управления также должна учитывать весь контекст. Как его станут использовать? Например, если он будет использоваться в условиях сильного холода, было бы разумно применять элементы управления, с которыми можно работать в громоздких перчатках. Кроме того, базовая технология может добавлять ограничения для интерфейса. Например, нельзя выполнять передачу приказа дрону более X раз в секунду.

Наконец, как такое можно запускать в производство? Возможно, принимавшие в этом участие проектировщики не были достаточно обучены и не получили должного руководства. Или этот дизайн так и не был утвержден. Если бы только кто-то – возможно, потенциальные пользователи – рассмотрел его, эти явные недостатки, вероятно, можно было бы исправить. Возможно, проектировщики создали хорошую модель, но в конце концов их план так и не был соблюден.

Независимо от качества, как только люди привыкнут к предмету и его интерфейсу, изменения нужно проводить с особой осторожностью. Если новая версия этого пульта дистанционного управления будет иметь совершенно иную организацию кнопок, пользователи, возможно, не захотят покупать новую версию, потому что им придется заново учиться работать с ним.

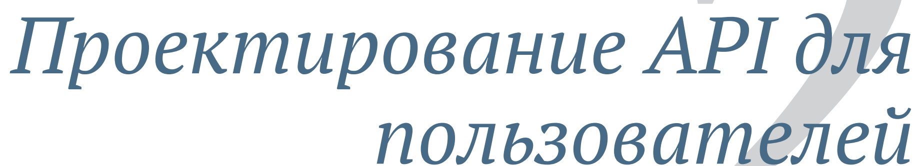
Как видите, проектирование интерфейса объекта требует сосредоточиться не только на кнопках. То же самое касается и проектирования API.

Проектирование API – это гораздо больше, чем просто проектирование простого и понятного интерфейса. Мы должны создать полностью безопасный интерфейс, ограничивая потребителям доступ к конфиденциальным данным или не давая им совершать лишние действия. Необ-

ходимо принимать во внимание весь контекст – каковы ограничения, как и кем будет использоваться API, как он создается и как может развиваться. Мы должны участвовать во всем жизненном цикле API – от первых обсуждений до проектирования, документирования, развития или вывода из эксплуатации. И поскольку компании обычно создают множество API, мы должны работать вместе с остальными проектировщиками, чтобы гарантировать, что все API организации имеют одинаковый внешний вид, чтобы создавать отдельные интерфейсы, которые были бы максимально согласованными, гарантируя таким образом, что все эти API так же просты для понимания и легки в использовании, как и каждый в отдельности.

## *Резюме*

- Веб-API превращают программное обеспечение в блоки многократного использования, которые можно использовать по сети с помощью протокола HTTP.
- API – это интерфейсы для разработчиков, которые создают приложения, использующие их.
- Дизайн API важен для всех API – открытых или закрытых.
- Плохо спроектированные API-интерфейсы могут быть использованы недостаточно, неправильно или вообще не использоваться, и даже небезопасны.
- Проектирование хорошего API требует, чтобы вы учитывали весь контекст приложения, а не только сам интерфейс.



# 2 Проектирование API для пользователей

**В этой главе вы узнаете:**

- на что обращать внимание при разработке API;
- подход к проектированию API, как при проектировании пользовательского интерфейса;
- как правильно определить реальные цели API.

Если вам не терпится ринуться в бой и приступить к проектированию программного интерфейса, вынужден вас огорчить: придется подождать до следующей главы. Здесь мы рассмотрим потребности пользователей API.

Когда вы хотите построить или создать что-то, вам нужен план. Вы должны определить, что вы хотите сделать, прежде чем приступить. Проектирование API не исключение.

API не предназначен для слепого раскрытия данных и возможностей. API, как и любой обычный пользовательский интерфейс, создан для его *пользователей*, чтобы помочь им добиться *своих целей*. В случае с API социальных сетей некоторые из этих целей могут заключаться в том, чтобы поделиться фото, добавить друга или показать список друзей. Эти цели формируют функциональный план, необходимый для проектирования эффективного API. Вот почему определение целей является решающим шагом в процессе проектирования API.

Цели – то, чего пользователи могут достичь с помощью вашего API – должны иметь смысл для пользователей, и ни одну из них нельзя упустить.

Определение релевантного и всеобъемлющего списка целей требует акцентирования внимания на *точке зрения пользователя*, а именно пользователей API и программного обеспечения, использующего API. Эта точка зрения является краеугольным камнем проектирования API, и именно ей должен руководствоваться проектировщик на протяжении всего процесса проектирования. Чтобы ее придерживаться, вам нужно не только понять ее, но и помнить о другой точке зрения, которая может помешать разработке API, – точке зрения поставщика (организации и программного обеспечения, открывающего доступ к API).

## 2.1 Правильная точка зрения для проектирования повседневных пользовательских интерфейсов

Проектировщики API должны многому научиться при проектировании обычных пользовательских интерфейсов, будь то интерфейсы физических или виртуальных объектов. От повседневных физических интерфейсов (двери, кухонные приборы или пульты дистанционного управления телевизора) до виртуальных (веб-сайты или мобильные приложения) – все они используют общие принципы проектирования, которые также должны применяться к проектированию API. Выбор правильной точки зрения является одним из наиболее важных аспектов проектирования всех этих интерфейсов и API.

**ПРИМЕЧАНИЕ.** Если вы сосредоточитесь на том, что происходит «под капотом», это приведет к полной катастрофе. Если сфокусироваться на том, что могут делать пользователи, – все пройдет гладко.

При проектировании API разделение этих двух точек зрения и их понимание поначалу не всегда дается легко. Но если перенести это в реальный мир, все становится очевидным, и то, как эти точки зрения могут повлиять на проектирование API, становится более понятным.

### 2.1.1 Когда вы фокусируетесь на том, как все работает, это приводит к возникновению сложных интерфейсов

Позвольте мне торжественно представить вам кухонный радар 3000 (показанный на рис. 2.1). Согласно рекламе, благодаря этому прибору «у вас на кухне появятся самые современные компоненты армейского образца, поэтому вы никогда не испортите ни одно блюдо и станете самым быстрым поваром в городе». Ого, звучит просто захватывающе, не правда ли? Ну, не совсем.

Что же такое кухонный радар 3000? Название ничего не говорит нам о его назначении. Может быть, панель управления даст нам какие-то подсказки… а может быть и нет.

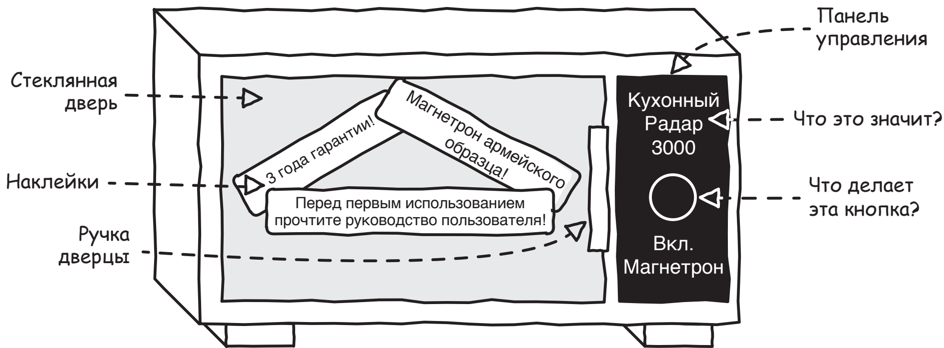



Рис. 2.1. Кухонный радар 3000

Как видно на рис. 2.1, тут есть одна кнопка – **Вкл. магнетрон**. Что такое магнетрон? Что происходит, если его включить? Это устройство – полная загадка.

Но посмотрите, на стеклянной двери есть наклейка, которая предлагает нам прочитать руководство пользователя (рис. 2.2). Может быть, мы найдем там полезную информацию, которая поможет нам понять назначение этого странного кухонного прибора и то, как его использовать.

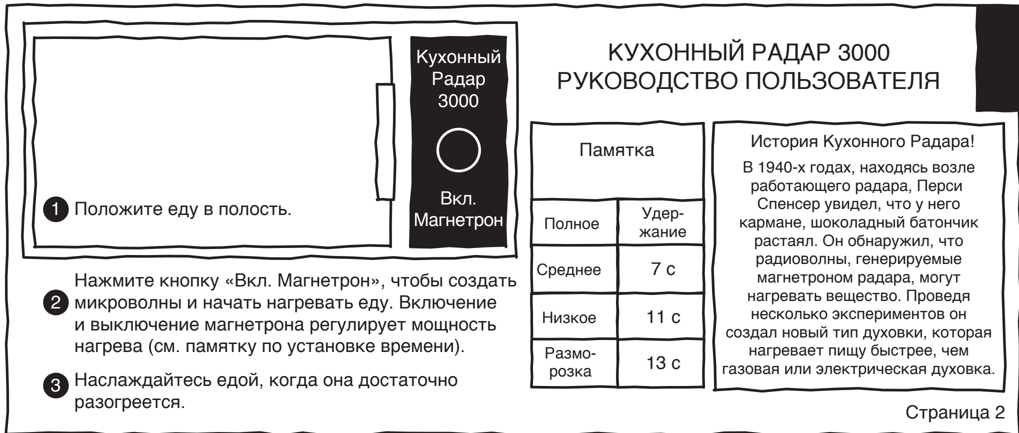



Рис. 2.2 (Безумное) руководство пользователя кухонного радара 3000

Итак, согласно руководству пользователя, кухонный радар 3000, похоже, представляет собой новый тип духовки, в которой для приготовления пищи используются радиомикроволны или просто микроволны. Он был изобретен кем-то, кто работал с радарами, отсюда и название. Что за ужасная идея: называть что-либо, основываясь на его истории, и полностью скрывать его истинное предназначение!

Итак, как он работает? Когда пользователи нажимают и удерживают кнопку **Вкл. магнетрон**, включается компонент под названием магнетрон. Работа этого компонента заключается в генерировании радиомикроволн.

Эти микроволны генерируют тепло, когда они проходят через пищу, помещенную в кухонный радар 3000. После того как еда приготовлена, пользователи могут отпустить кнопку, чтобы выключить магнетрон.

Эта панель управления не очень удобна – пользователи должны сами измерять время, нажимая кнопку! А ведь именно тогда им нужно использовать всю мощность нагрева духовки. Что произойдет, если они захотят использовать только часть этой мощности? Они должны нажать и отпустить кнопку включения магнетрона в определенном темпе в соответствии с желаемой мощностью нагрева. Таким образом, магнетрон будет включаться и выключаться, генерируя меньше микроволн и, следовательно, меньше нагревая, чем если бы вы непрерывно удерживали кнопку. Это совершенное безумие! Кто бы стал охотно использовать такое устройство? Наверное, никто.

Откровенно говоря, это устройство на самом деле не было спроектировано. Оно только показывает свое внутреннее устройство (магнетрон) и историю (радар) своим пользователям. Его назначение трудно расшифровать. Мало того, что пользователи должны знать, как работает магнетрон, так еще и предоставляемая панель управления – просто кошмар. Похоже, это тот случай, когда проектировщик, сосредоточившись на том, как все работает, создал устройство, которое не только сложно понять, но и тяжело использовать. Можно ли было избежать этого? Конечно, да! Давайте посмотрим, как можно исправить эту катастрофу.

### 2.1.2 Когда вы фокусируетесь на том, что могут делать пользователи, это приводит к появлению простых интерфейсов

Как же усовершенствовать кухонный радар 3000, чтобы с ним было проще разобраться и проще использовать? Что же, если это устройство было спроектировано с упором на то, как оно работает, как насчет попытки перепроектировать его, думая об обратном? Давайте сосредоточимся на потребностях пользователей и соответствующим образом изменим дизайн панели управления этого устройства.

Этот кухонный прибор в основном представляет собой что-то вроде духовки. А чего хотят люди, когда они включают духовку, независимо от используемой технологии? Они хотят разогреть или приготовить еду. Можно было бы просто переименовать кухонный радар 3000 в микроволновую печь и заменить название кнопки **Вкл. магнетрон** на **Нагрев** или **Готовить**, как показано на рис. 2.3.

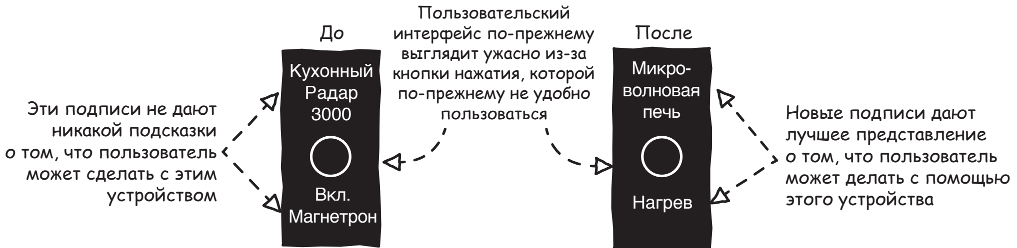



Рис. 2.3. Переименовываем кухонный радар 3000 в микроволновую печь

Ну что же, так лучше. Просто, изменив некоторые названия, мы упростили пользователям задачу, и теперь им легче понять, для чего предназначено это устройство: очевидно, что печь нужна для нагрева чего-либо, а кнопка запускает процесс.

Однако опыт использования этой микроволновой печи по-прежнему ужасен, особенно когда дело доходит до разогрева пищи на долю от полной мощности нагрева. Пользователям все еще приходится справляться с темпом нажатия/отпускания кнопки, чтобы разогревать что-либо медленнее. На рис. 2.4 показано, как это можно упростить.

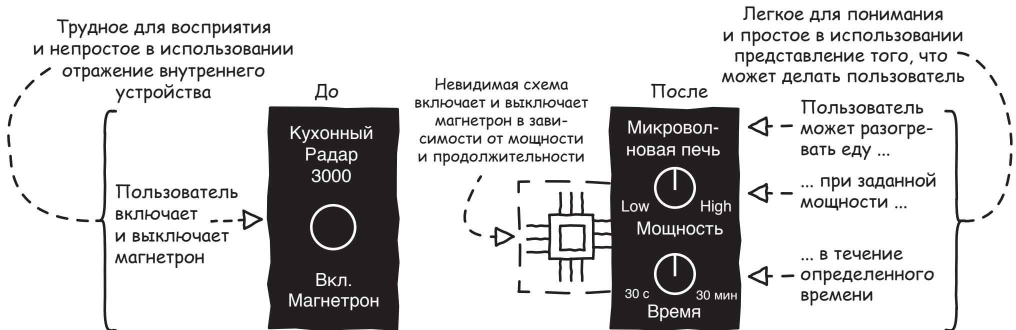



Рис. 2.4. Упрощаем процесс использования за счет изменения дизайна панели управления

Как люди обычно разогревают пищу с помощью духовки или другого устройства для приготовления пищи? Они нагревают ее в течение заданного *времени* при заданной *мощности*. Можно заменить кнопку **Нагрев** элементами управления, которые позволяют пользователям указывать продолжительность в секундах и минутах, а также выбирать низкую, среднюю или высокую мощность нагрева. Некие схемы за панелью управления будут обрабатывать магнетрон и включать и выключать его в соответствии с данными, введенными пользователем.

Превосходно! Теперь этот кухонный прибор легко понять и использовать благодаря новой панели управления. Новый интерфейс больше не является загадочным прямым доступом к внутренней работе устройства. Нет никакого намека на то, что в действительности происходит внутри; он фокусируется только на потребностях пользователя. Это наглядное представление того, чего можно достичь с помощью духовки.

Детали внутренней работы скрыты внутри схемы за панелью управления и не будут беспокоить пользователя.

Мы только что переделали панель управления кухонного прибора – ну и что? Как все это связано с дизайном API? Напрямую! Мы создали интерфейс, который является простым для понимания и простым в использовании представлением о том, что пользователи могут делать с устройством, не беспокоя себя неуместными проблемами, связанными с внутренним устройством. Если вы сохраните этот настрой при создании API, то станете отличным проектировщиком! Почему? Потому что API – это в основном панель управления для программного обеспечения, и он должен подчиняться тем же правилам, что и любой обычный интерфейс.

## 2.2 Проектирование интерфейсов программного обеспечения

API – это надписи и кнопки программного обеспечения, его панель управления. Это можно понять с первого взгляда, а можно не разгадать даже после нескольких дней исследований. Он может быть настоящим удовольствием или сплошным мучением, как и в случае с предметами из реальной жизни.

Создание API, который является простым в использовании и у которого понятный интерфейс, требует от нас проектирования с ориентацией на подходящую точку зрения: что могут делать пользователи. Если вы сосредоточитесь на том, как работает программное обеспечение, это приведет к полной катастрофе. Если сфокусироваться на том, что с ним могут делать пользователи, все пройдет гладко – как и при проектировании объекта из реальной жизни.

### 2.2.1 API как панель управления программным обеспечением

Когда вы используете микроволновую печь или любой другой предмет повседневного пользования, вы взаимодействуете с ним через панель управления, его интерфейс. Вы читаете надписи, нажимаете кнопки или поворачиваете ручки.

То же самое происходит, когда вы хотите взаимодействовать с программным обеспечением программными средствами: вы используете его API. Как уже упоминалось, API – это панель управления программным обеспечением. Но что конкретно это значит? Давайте превратим наш переделанный кухонный радар 3000, нашу микроволновую печь, в программное обеспечение (как показано на рис. 2.5), чтобы ответить на этот вопрос.

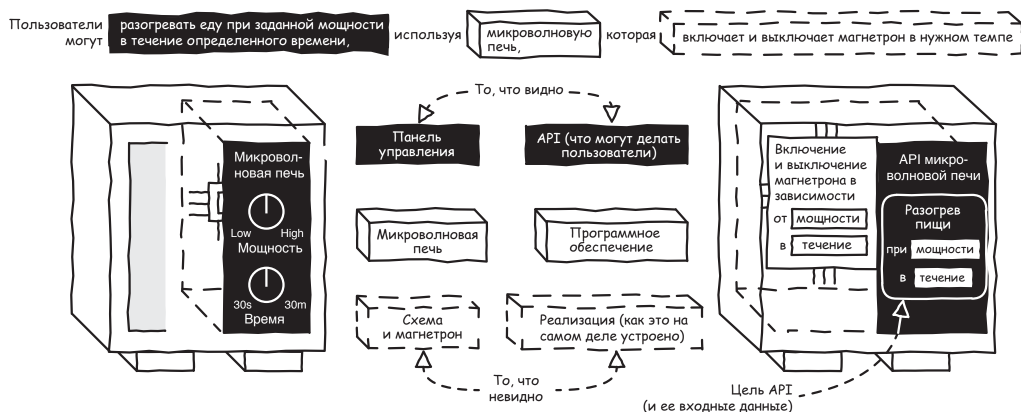



Рис. 2.5. Сравнение микроволновой печи с программным обеспечением

Вся микроволновая печь становится программным обеспечением, панель управления – API программного обеспечения, а схема, стоящая за панелью управления, – реализацией.

API – это то, что видят пользователи; представление того, что они могут сделать. Реализация – это код, работающий позади API. Это то, как на самом деле все делается, но остается невидимым для пользователя.

API обеспечивает представление целей, которых можно добиться с его помощью. API микроволновой печи позволяет пользователям греть пищу. Для достижения цели может потребоваться некая информация (входные данные). Пользователи должны указать параметры мощности и продолжительность нагрева пищи. Для реализации цели используется информация, предоставленная через API. В данном случае реализация включает и выключает магнетрон в заданном темпе в соответствии с предоставленной мощностью в течение предоставленной продолжительности. И когда цель достигнута, можно получить какую-то информацию.

Итак, API – это панель управления, созданная для программного взаимодействия с программным обеспечением для достижения целей. Но, как и в обычных интерфейсах, эти цели могут быть выражены с точки зрения поставщика (включить магнетрон), а не потребителя (разогреть еду). Для API это такая же серьезная проблема, как и для повседневных интерфейсов?

### 2.2.2 Ориентация на точку зрения потребителя для создания простых API

Чтобы понять последствия проектирования API с точки зрения поставщика (неоправданно открывая доступ к внутреннему устройству), давайте сравним псевдокод, необходимый для использования API-интерфейса кухонного радара 3000 и API микроволновой печи.

API-интерфейс кухонного радара 3000 предлагает две цели: включить магнетрон и выключить его. В приведенном ниже листинге показано, как использовать эти элементы управления для разогрева пищи.

**Листинг 2.1. Использование API кухонного радара 3000**

```
if <power> is high ①
   turn magnetron on
   wait for <duration>
   turn magnetron off
else
   if <power> is medium ②
      cycle = 7s
   else if <power> is low
      cycle = 11s
   else if <power> is thaw
      cycle = 13s
   end if
   for <duration> ③
      turn magnetron on
      wait for <cycle>
      turn magnetron off
   end for
end if
```

① **При высокой мощности цикла включения/выключения нет.**

② **Расчет времени ожидания цикла включения/выключения на основе мощности.**

③ **Чередование цикла включения/выключения.**

Если разработчики хотят создать программу для разогрева пищи с использованием этого API, они должны будут включить магнетрон, подождать определенное время (продолжительность), а затем выключить его. Но когда они хотят использовать полную мощность нагрева, это простой случай использования. Если они хотят нагревать что-то с малой долей полной мощности, им придется включать и выключать магнетрон с такой скоростью, чтобы достичь желаемой величины нагрева.

Использование API кухонного радара 3000, как и его реального аналога, обернется кошмаром. Разработчики должны написать сложный код для своего программного обеспечения, чтобы использовать его. Они даже могут сделать ошибки. Кстати, в этом псевдокоде есть ошибка. Вы это заметили? В приведенном ниже листинге показано, как ее исправить.

**Листинг 2.2. Исправляем ошибку**

```
// Чередование цикла включения/выключения;
for <duration>
   turn magnetron on
   wait for <cycle>
   turn magnetron off
```

```
  wait for <cycle> ①
end for
```

① **Этой строки не было**

В цикле пропущена строка `wait for <cycle>`. Это, безусловно, не самый лучший код, но он работает. Давайте теперь посмотрим, как сделать то же самое с помощью API микроволновой печи, которая обеспечивает единственную цель: `heat food`. Таким образом, разогреть пищу будет просто, как показано в приведенном ниже листинге.

**Листинг 2.3. Использование API микроволновой печи**

```
heat food at <power> for <duration>
```

Разработчикам нужно написать только одну строку псевдокода; что может быть проще. Будучи разработчиком, который хочет предоставить программное обеспечение для разогрева пищи, какой API вы бы предпочли использовать: тот, что нуждается в сложном, подверженном ошибкам коде, или тот, которому нужна лишь одна защищенная от ошибок строка кода? Конечно же, это риторический вопрос.

Как и панель управления, сложность или простота API зависит, прежде всего, от точки зрения, на которую вы ориентируетесь при разработке. Было ли вам известно что-нибудь о магнетронах и изобретении микроволновой печи, прежде чем вы начали читать эту книгу? Вероятно, нет. Почему? Потому что панель управления микроволновой печи разработана с точки зрения пользователей. Она не предоставляет доступ к не относящемуся к делу внутреннему устройству и не требует, чтобы пользователь был экспертом в области магнетронов или радаров. Ее может взять любой, кто хочет разогреть еду. А помешал ли этот пробел в знаниях использовать микроволновую печь? Абсолютно нет. Почему? Потому что, когда вы используете ее, вы просто хотите разогреть еду. Вас не волнует, как именно это будет сделано.

Вот как нужно проектировать любой API – с точки зрения потребителя, а не поставщика. На рис. 2.6 показаны две эти точки зрения.

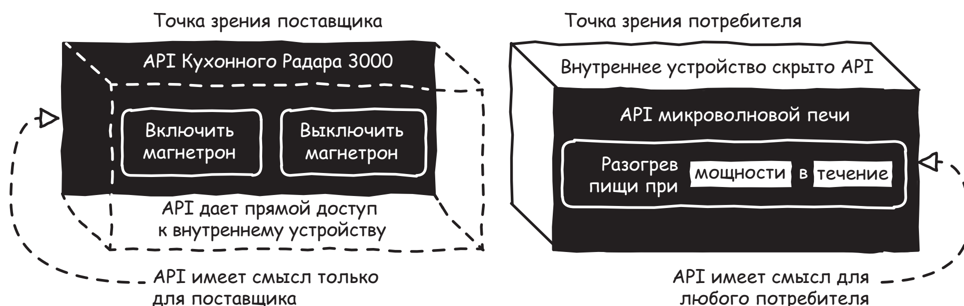



**Рис. 2.6. Точка зрения потребителя и поставщика**

API, спроектированный с точки зрения поставщика, представляет собой лишь окно, показывающее внутреннюю работу и, следовательно,

представляющее цели, которые имеют смысл только для поставщика. Как вы видели в примере с API-интерфейсом кухонного радара 3000, такой API неизбежно будет сложным в использовании, и потребители не смогут просто делать то, что хотят. Напротив, API, спроектированный с точки зрения потребителя, представляет собой экран дисплея, скрывающий внутреннее устройство. Он показывает только те цели, которые имеют смысл для любого потребителя, и позволяет ему просто достичь того, чего он хочет. Сосредоточив внимание на этой точке зрения, API будет на правильном пути, чтобы гарантировать удобство в использовании.

Хорошо, при проектировании API мы должны в первую очередь думать о потребителе. И таким образом, представляется важным четко определить цели, которые потребители могут достичь при использовании API, чтобы обеспечить создание простого для понимания и несложного в применении API. Но как сделать это правильно и исчерпывающе?

## 2.3 Определение целей API

Основываясь на нашем эксперименте с микроволновой печью, я надеюсь, теперь вы убеждены, что самый первый шаг в процессе проектирования API-интерфейса – это установить, чего могут добиться его пользователи, – определить реальные цели API-интерфейса. В случае с нашим API микроволновой печи это был просто подогрев пищи; но для API социальных сетей это могут быть такие цели, как обмен фотографиями, добавление друга или получение списка друзей. Однако такие простые описания недостаточно точны. Как именно пользователь добавляет друга? Что для этого нужно? Что пользователь получает взамен? При разработке API важно иметь глубокие и точные знания о том,

- кто может использовать API;
- что он может делать;
- как он это делает;
- что ему нужно для этого;
- что он получает взамен.

Эти фрагменты информации являются основой вашего API. Они нужны вам для проектирования правильного программного интерфейса. Метод и цели API, описанные в этой главе, являются простыми, но мощными инструментами, которые помогут вам получить всю необходимую информацию.

Цель проектирования программного обеспечения (или чего-либо еще), которое удовлетворяет потребности пользователей, не является чем-то новым. Это всегда было целью, сколько существует программное обеспечение. Многочисленные методы существовали и по-прежнему изобретаются, для того чтобы коллекционировать потребности пользователей, глубже и точнее понимать их и наконец создать программное обеспечение, которое более или менее эффективно и точно выполняет свои задачи. Все эти методы могут использоваться для определения целей API. Не

стесняйтесь использовать метод, который вам знаком, или адаптировать тот, что представлен в этой книге, по своему усмотрению, если вы знаете, кто ваши пользователи, что они могут делать и как они могут это сделать. Эти принципы описываются в последующих разделах.

### 2.3.1 Отвечая на вопросы «что?» и «как?»

Когда мы переделали Кухонный радар 3000 в микроволновую печь, мы ответили на два вопроса, как показано на рис. 2.7. Первый вопрос: «Чего хотят люди, когда они используют духовку?» Ответ: «Они хотят разогревать пищу». Этот ответ привел к появлению второго вопроса: «Как люди разогревают пищу?» Ответ: «Они разогревают пищу с заданной мощностью в течение определенного периода времени». Этот скромный опрос помог нам определить информацию, необходимую для проектирования простой и удобной для пользователя цели для API микроволновой печи, заключающейся в разогреве пищи, – *что* и *как*. Этот пример показывает два основных вопроса, задаваемых при определении списка целей API:

- Что могут делать пользователи?
- Как они это делают?

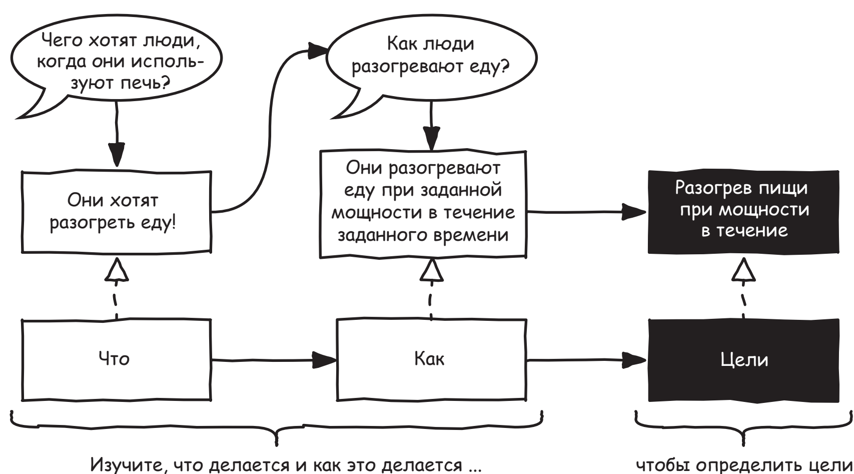



**Рис. 2.7. Как мы переделали панель управления кухонного радара 3000**

С помощью этих вопросов мы примерно описываем, что можно сделать с помощью API (*что*), и разбиваем их на этапы (*как*), при этом каждый этап становится целью API.

Но в этом примере «что» и «как» – в принципе, одно и то же. Что, нам действительно нужно выполнять это разбиение? Определенно. К сожалению, пример с микроволновой печью слишком упрощен, чтобы проиллюстрировать это. Нам нужен более сложный вариант использования, поэтому давайте поработаем над API сайта онлайн-магазина или мобильного приложения. Каков тип действий и способ их реализации в случае с этим API? Все показано на рис. 2.8.

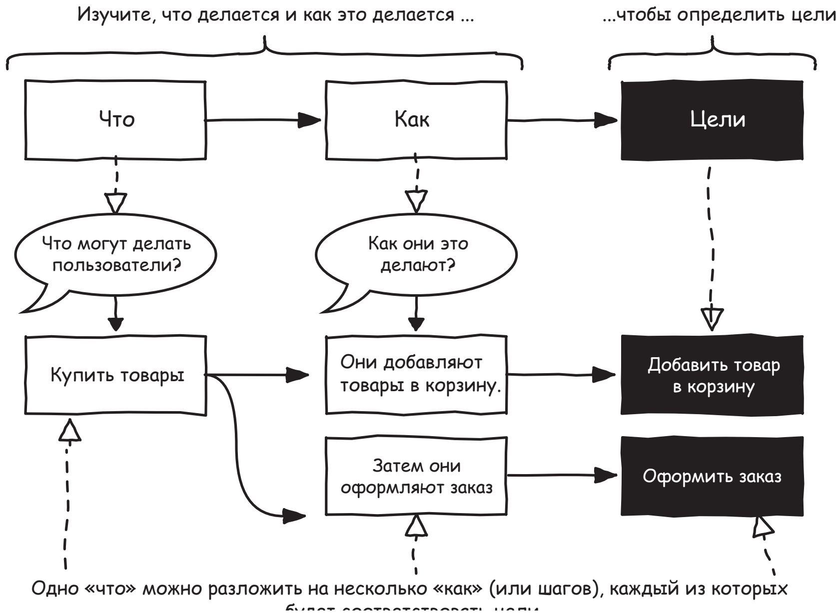



**Рис. 2.8.** API онлайн-магазина

Что люди делают, когда совершают покупки в интернете? Покупают товары. А как они их покупают? Добавляют их в свою корзину, а затем оформляют заказ.

Таким образом, покупка товаров представляет собой процесс, состоящий из двух этапов, который можно представить с помощью следующих целей: добавить товар в корзину и оформить заказ. Без разбиения мы могли бы ошибочно перечислить цель покупки одного товара. Вот почему следует разложить действия на способы их реализации – если мы этого не сделаем, то можем пропустить какие-то цели.

Отлично, мы закончили! Чтобы определить цели API, просто нужно приблизительно перечислить, что пользователи могут делать, и разбить эти действия на этапы, изучив как они их совершают. Каждый этап соответствует цели. Итак, давайте сделаем следующий шаг и спроектируем программный интерфейс, соответствующий этим целям. Но подождите... думаю, мы что-то упустили.

Переделав микроволновую печь, мы не просто определили цель, которая состоит в разогревании пищи. Мы неявно определили мощность и продолжительность в качестве входных данных, предоставляемых пользователем. Это помогло нам переделать панель управления кухонного радара 3000, поэтому было бы интересно определить входные и выходные данные цели.

### 2.3.2 Определение входных и выходных данных

Для достижения цели могут потребоваться входные данные. В случае с нашим API микроволновой печи входными данными были мощность и длительность. Эти данные помогли нам спроектировать панель управ-

ления и API микроволновой печи. Цель может даже вернуть выходные данные, когда она будет достигнута. Эти выходные данные также влияют на дизайн API. Итак, давайте определим эти цели для нашего API онлайн-магазина, как показано на рис. 2.9.

**Рис. 2.9. Действия, способы их реализации, а также входные и выходные данные**

Что люди делают, когда совершают покупки в интернете? Покупают какие-то товары. А как они их покупают? Добавляют их в свою корзину, а затем оформляют заказ.

Что касается этих двух первых вопросов, здесь нет ничего нового. Давайте теперь подробно рассмотрим каждый шаг, чтобы определить его входные и выходные данные.

Начнем с добавления товаров в корзину. Что нужно людям, чтобы это сделать? Очевидно, им нужен товар и корзина. Хорошо, а они получают что-то взамен, когда добавляют товар в свою корзину? Нет. Эти новые вопросы, похоже, не дают нам действительно полезной информации; ответы довольно очевидны. Может быть, мы получим что-то более интересное на этапе оплаты.

Нужно ли людям что-нибудь, чтобы оформить заказ, отправленный в корзину? Ясное дело, корзина. Ответ по-прежнему очевиден. А они получают что-то взамен? Да, подтверждение заказа. (Этот ответ был не так очевиден.)

Можно ли было догадаться, что API будет манипулировать заказами, просто взглянув на корзину покупок? Возможно, да, если API такой простой, но, возможно, и нет, если API более сложный.

Чтобы спроектировать правильную панель управления программным обеспечением, содержащую все необходимые кнопки и надписи, нам необходимо иметь четкое представление не только о целях, но и о том, что нужно для их достижения и что мы получаем взамен. Идентифика-

ция целей API заключается не только в том, что с ним можно сделать, но и в том, какими данными можно манипулировать с его помощью. Вот почему нам нужно добавить в наш список еще два вопроса:

- Что могут делать пользователи?
- Как они это делают?
- Новый вопрос для определения входных данных: что им нужно для этого?
- Новый вопрос для определения выходных данных: что они получают взамен?

Отлично, теперь у нас и правда все готово! Давайте перейдем к следующему шагу и спроектируем программный интерфейс, соответствующий этим целям. Но подождите... мы снова что-то упустили! Как пользователь получает товар, чтобы добавить его в корзину? Кажется, это очевидный вопрос может представлять интерес при обнаружении некоторых пропущенных целей.

### 2.3.3 Выявляем недостающие цели

Как пользователь получает товар, чтобы добавить его в корзину? Это загадка. Кажется, мы упустили одну из целей, а может и больше. Как избежать этого? Универсального средства от этого не существует, но, изучая источники входных данных и использование выходных данных, можно обнаружить пропущенные «что» или «как», а следовательно, пропущенные или неопознанные цели.

Итак, как пользователи получают товар, чтобы добавить его в корзину? Вероятно, сначала они ищут его по названию или описанию. Таким образом, мы можем добавить новый шаг «Поиск товаров» в список действий «Покупка товаров». Мы не должны забывать применить свой опрос к этому новому шагу, как показано на рис. 2.10.

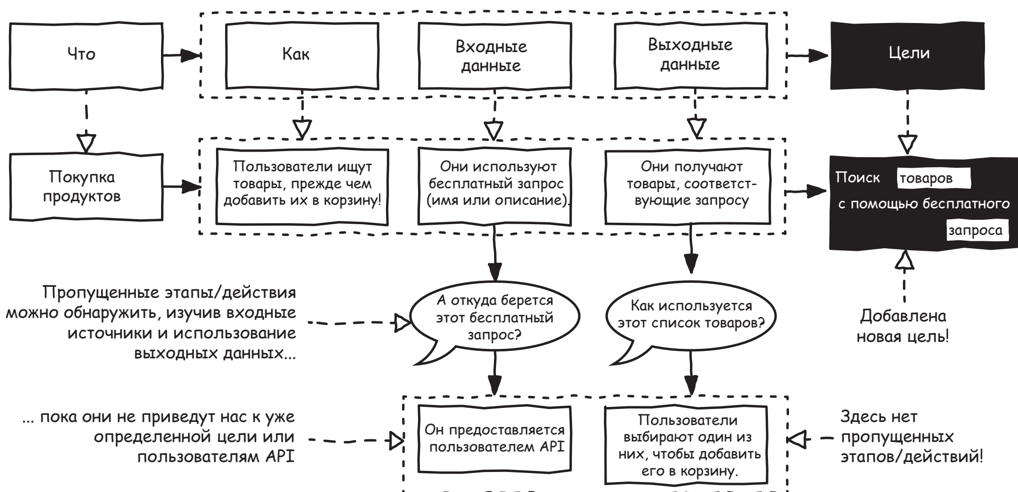



Рис. 2.10. Добавление отсутствующего способа реализации действия

Что нужно пользователям для поиска товара? Бесплатный текстовый запрос; это может быть имя или описание, например. Что возвращается в результате этого поиска? Список товаров, соответствующих запросу. Как пользователи получают поисковый запрос? Они сами предоставляют его. Как применяется список товаров? Пользователи выбирают один из них, чтобы добавить его в корзину.

Мы покопались во всех входных и выходных данных, пока ответы не привели нас к пользователям API или к уже определенной цели. Таким образом, мы уверены, что на этом пути больше нет пропущенных этапов. Отлично. Одна проблема решена. Давайте рассмотрим следующий шаг, «Оформление заказа», как показано на рис. 2.11.

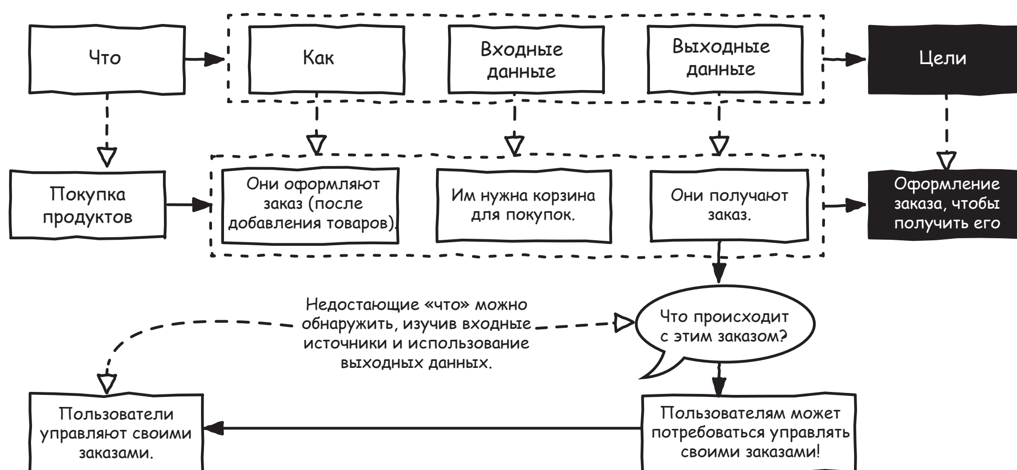



**Рис. 2.11. Добавление недостающего действия**

Здесь требуется корзина и возвращается заказ. Мы не будем выяснять, откуда взялась корзина (вы можете сделать это в качестве упражнения), но что делать с заказом? Почему он возвращается пользователю? Может быть, таким образом пользователь способен проверить статус заказа? Интересно, но я чувствую, что это еще не все. Пользователи, вероятно, должны будут управлять своими заказами. Я думаю, мы только что заметили недостающее действие! Итак, давайте добавим новый ответ «Они управляют заказами» на наш вопрос «Что могут делать пользователи?» и приступим к расследованию.

Как пользователи управляют своими заказами? Они должны иметь возможность перечислять свои заказы в хронологическом порядке, чтобы проверить их статус. Итак, у нас есть два этапа для изучения. Получившийся список целей API онлайн-магазина показан на рис. 2.12.

**Рис. 2.12.** **Список целей API онлайн-магазина с новым действием «Пользователи управляют своими заказами»**

Во-первых, список заказов: какие входные данные необходимы для составления списка заказов? Никаких. Что возвращается? Список заказов. Откуда поступают входные данные? Нам не нужны никакие входные данные для составления списка заказов, поэтому нам не нужно думать о том, откуда они берутся. Что мы делаем с выходными данными? Пользователи могут выбрать один из заказов в списке, чтобы проверить его статус. На этом все, что касается первого этапа управления заказами.

Второй этап, проверка статуса заказа: какие входные данные необходимы для проверки статуса заказа? Заказ. Что возвращается? Статус заказа. Откуда поступает заказ? Из корзины или списка заказов. Что мы делаем со статусом заказа? Нам нужно предоставить эти данные, чтобы проинформировать пользователей, не более того.

Фантастика. Новое действие, еще две способа реализации, и цели определены. Итак, давайте расширим наш опрос, добавив еще два вопроса, чтобы определить источники входных данных и использование выходных данных:

- Что могут делать пользователи?
- Как они это делают?
- Что им нужно для этого?
- Что они получают взамен?
- Новый вопрос для определения недостающих целей: откуда поступают входные данные?

- Новый вопрос для определения недостающих целей: как используются выходные данные?

Изучение источников ввода и использования выходных данных определенно помогает определить недостающие цели API. Но мы по-прежнему не готовы к проектированию программного интерфейса. Список целей все еще неполон, потому что я намеренно совершил еще одну ошибку. Знаете какую? Если вы примените наш опрос к нашему последнему списку целей, то сможете найти ее. По правде говоря, пропущенных целей много; мы лишь поверхностно затронули эту тему. Но я имею в виду товары, возвращаемые в результате поиска. Этот ответ рассматривается в следующем разделе.

### 2.3.4 Идентификация всех пользователей

Мы уже упомянули, что пользователи могут искать товары и добавлять их в свою корзину, но откуда берутся эти товары? Конечно же, из каталога! Однако эти товары не появляются волшебным образом сами по себе в этом каталоге. Должно быть, кто-то поместил их туда. Будучи клиентом, вы не добавляете товары в каталог самостоятельно; это делает какой-то администратор. Понимаете?

Просто применив нашу обычную линию вопросов, мы снова обнаружили дыру в нашем списке целей API онлайн-магазина. Здорово! Но, вместо того чтобы ждать, когда эти пользователи будут обнаружены путем изучения входных и выходных данных, можно было бы действовать более эффективно, добавив в наши вопросы новое измерение, как показано на рис. 2.13.

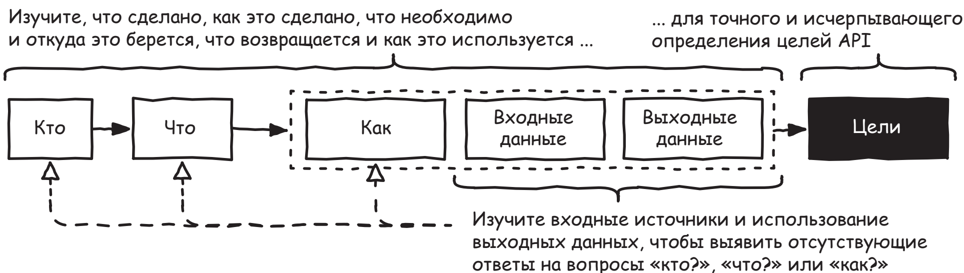



**Рис. 2.13.** Изучение пользователей, действий, способов их реализации, входных и выходных данных для определения целей

Идентификация различных типов пользователей является обязательной при построении исчерпывающего списка целей API. Таким образом, мы должны добавить еще один запрос к нашей линии вопросов, чтобы явно идентифицировать их все:

- Новый вопрос для идентификации всех пользователей и исключения пропущенных действий: кто такие пользователи?
- Что они могут делать?
- Как они это делают?
- Что им нужно для этого?

- Что они получают взамен?
- Откуда поступают входные данные?
- Как используются выходные данные?

Если *сначала* мы определим различные типы пользователей нашего API, нам будет проще составить полный список целей. Обратите внимание, что термин *пользователь* используется здесь в широком смысле; это может быть конечный пользователь, использующий приложение, которое потребляет API, само потребительское приложение или роли или профили конечного пользователя или потребительского приложения. И помните, мы по-прежнему можем полагаться на изучение входных и выходных данных, чтобы убедиться, что мы обнаружили всех пользователей.

Тут возникает множество вопросов для решения. Давайте посмотрим, как можно упростить этот список вопросов с помощью таблицы целей API.

### 2.3.5 Использование таблицы целей API

Теперь, когда мы знаем, какие вопросы задавать и почему мы должны их задавать, чтобы определить исчерпывающий и точный список целей API, давайте посмотрим, как справиться с этим процессом с помощью таблицы целей API, показанной на рис. 2.14.

*Таблица целей API* состоит из шести столбцов, соответствующих процессу, который мы обнаружили в предыдущих разделах:

- *Кто* – здесь вы перечисляете пользователей API (или профили);
- *Что* – здесь вы перечисляете, что могут делать эти пользователи;
- *Как* – здесь вы разбиваете каждое действие на этапы;
- *Входные данные (источник)* – здесь вы перечисляете, что необходимо для каждого шага и откуда это берется (чтобы определить недостающих пользователей, действия или способы их реализации);
- *Выходные данные (использование)* – здесь вы перечисляете, что возвращает каждый этап и как это используется (чтобы определить недостающих пользователей, действия или способы их реализации);
- *Цели* – здесь вы четко и кратко переформулируете каждый способ реализации + входные данные + выходные данные.

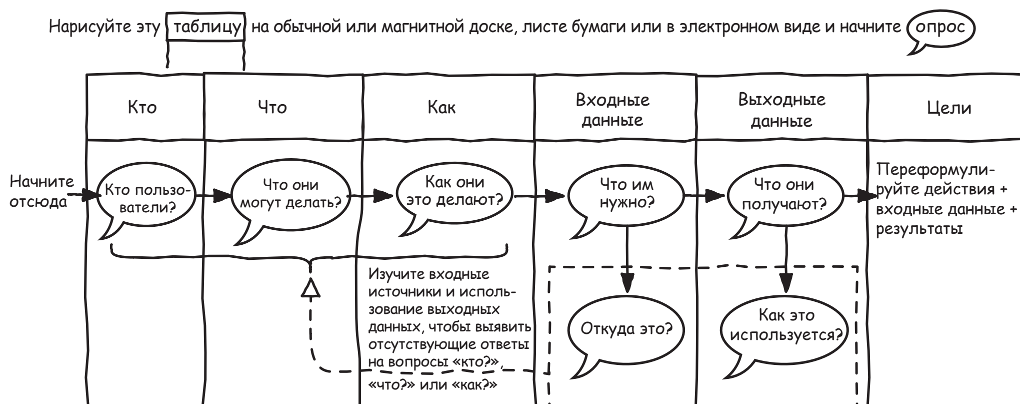



Рис. 2.14. Таблица целей API

| Кто | Что | Как | Входные данные | Выходные данные | Цели |
|---|---|---|---|---|---|
| Клиенты | Покупка товаров | Поиск товаров | Каталог (управление каталогом), бесплатный запрос (предоставляется пользователем) | Товары (добавить товар в корзину) | Поиск товаров в каталоге с помощью бесплатного запроса |
| | | Добавить товар в корзину | Товар (поиск товаров) корзина (принадлежит пользователю) | | Добавить товар в корзину |
| Администратор | Управление каталогом | Добавить товар в каталог | Каталог (принадлежит пользователю), товар (предоставляется пользователем) | | Добавить товар в каталог |

Рис. 2.15. Таблица целей API онлайн-магазина (частичное представление)

На рис. 2.15 показано частичное представление таблицы целей нашего API онлайн-магазина. Таблица целей API и лежащий в ее основе метод опроса помогут вам представить, кто использует API, что они могут с ним делать, как они это делают, что им нужно и что они получают взамен. Это фундаментальная информация, необходимая для проектирования программного интерфейса, представляющего определенные цели.

Вы, возможно, заметили, что мы не говорили о мелкоструктурных данных и ошибках. Мы поговорим о них позже в главах 3 (раздел 3.3) и 5 (раздел 5.2). Таблица целей API – это только высокоуровневое представление; на этом этапе не следует слишком углубляться в детали.

Имейте в виду, что, даже не вдаваясь в такие подробности, наполнение целей API может быть довольно сложным в сложных контекстах. Пользователей или профилей может быть много или слишком много вариантов использования. Это не ограничивается проектированием API и происходит при проектировании любого программного решения. Не

пытайтесь охватить все случаи использования одним выстрелом. Вместо этого сфокусируйтесь на небольшом наборе вариантов. Если действие содержит множество этапов или ответвлений, сосредоточьтесь на главном пути, а после этого проверьте, есть ли изменения, приводящие к появлению новых целей в других путях. То же самое касается пользователей: попытка изучить все действия для всех пользователей или профилей может быть непростой. Сосредоточьтесь на основном пользователе или профиле и после этого проверьте, если есть варианты для других.

Перечисление целей API – итерационный процесс. Вы должны действовать шаг за шагом – не пытаясь сделать все сразу. И вам также нужно будет уточнить и изменить этот список на основе соображений или ограничений, таких как удобство использования, производительность или безопасность. Вы узнаете о них в ходе прочтения этой книги.

**ПРИМЕЧАНИЕ.** Не стесняйтесь адаптировать этот метод и инструмент или использовать любой другой метод, с которым вы знакомы, при условии, что он позволяет вам получать информацию, перечисленную в таблице целей API.

К сожалению, данный метод не гарантирует, что ваш список целей API будет определен с точки зрения потребителя. Да, первый вопрос («Кто такие пользователи?») не мешает точке зрения поставщика появиться в целях вашего API. Чтобы быть уверенным, что вы не попадете в одну из этих ловушек при создании списка целей API, нам нужно изучить различные аспекты предательской точки зрения поставщика.

## 2.4 Избегаем точки зрения поставщика при проектировании API

Независимо от того, проектируете ли вы свой API с нуля или опираетесь на существующие системы, точка зрения поставщика неизбежно будет проявляться на каждом этапе его проектирования. Знание различных аспектов является основополагающим для любого проектировщика API, который надеется остаться на тропе, ведущей к точке зрения потребителя, и создавать легкие для понимания и простые в использовании API.

Помните API нашего кухонного радара 3000? Его название, придуманное на основе предыдущих событий, и его недружественные по отношению к пользователю функции включения и выключения магнетрона. Предоставление доступа к его внутреннему устройству – вопиющий пример проектирования API, находящейся под сильным влиянием точки зрения поставщика. К сожалению, эта точка зрения не всегда так очевидна, но среди людей, занимающихся проектированием программного обеспечения, существует поговорка, которая выявляет темные стороны точки зрения поставщика. Она известна как *закон Конвея*, и его часто цитируют, чтобы объяснить, как дизайн системы может зависеть от ее внутренней работы. Этот закон гласит:

*«Организации, проектирующие системы, ограничены дизайном, который копирует структуру коммуникации в этой организации».*

*Мел Конвей*
*«How Do Committees Invent?»*
*Журнал Datamation, апрель 1968 года*

Эту поговорку можно применить к широкому кругу систем – от общественных организаций до программных систем и, конечно, API. Она означает, что на дизайн API может влиять структура коммуникации организации, предоставляющей его, как показано на рис. 2.16.

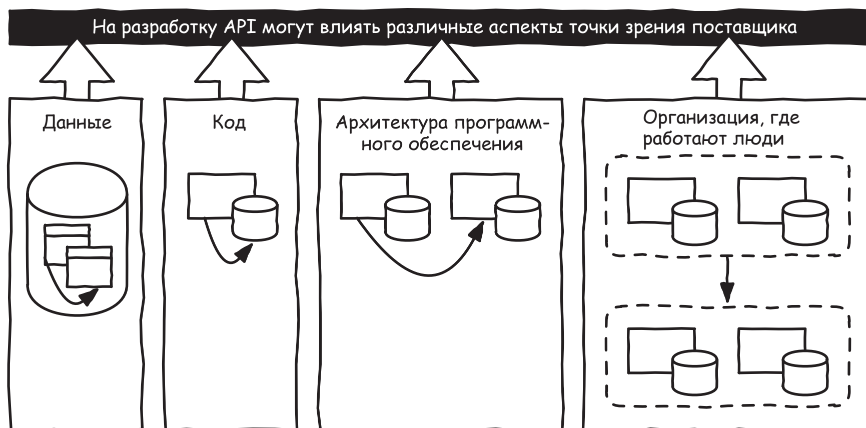



**Рис. 2.16. Различные аспекты точки зрения поставщика**

Данные, код и бизнес-логика, архитектура программного обеспечения и общественная организация формируют коммуникационную структуру компании и, следовательно, могут влиять на проектирование ее API.

### 2.4.1 Как избежать влияния данных

По сути, API – это способ обмена данными между двумя частями программного обеспечения – потребителем и поставщиком. Поэтому, к сожалению, часто встречаются конструкции API, которые отражают основную организацию данных. То, как данные структурированы или названы, может влиять на проектирование API, как показано на рис. 2.17.

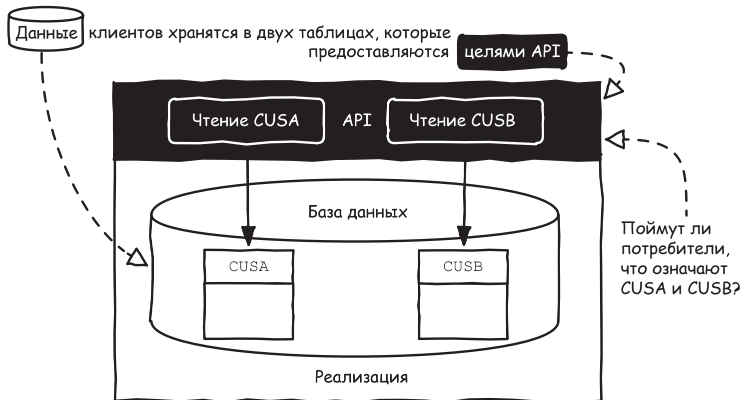



**Рис. 2.17.** Представление доступа к организации данных через API

Допустим, наша реализация API онлайн-магазина обрабатывает информацию о клиентах в двух таблицах под названием CUSA и CUSB (не спрашивайте, откуда такое название!). Дизайн API, на который влияет такая структура данных, может напрямую открыть две цели: чтение CUSA и чтение CUSB. Такой API трудно понять. Что именно означают CUSA и CUSB? (я также позволю вам представить себе загадочные имена столбцов таблиц, доступ к которым может быть открыт потребителю напрямую). Кроме того, он сложен в использовании! Потребители должны использовать две цели для получения всех клиентских данных.

**ВНИМАНИЕ!** Если список целей и данные вашего API слишком совпадают с вашей базой данных – будь то по структуре или названию, – возможно, вы проектируете свой API с точки зрения поставщика. В этом случае не стесняйтесь перепроверить, действительно ли пользователям API нужно иметь доступ к таким деталям.

Откровенно говоря, предоставление доступа к вашей модели базы данных чаще всего является ужасной идеей и может привести к малоприятному взаимодействию с пользователем. К счастью, на рис. 2.18 показано, как можно решить эту проблему.

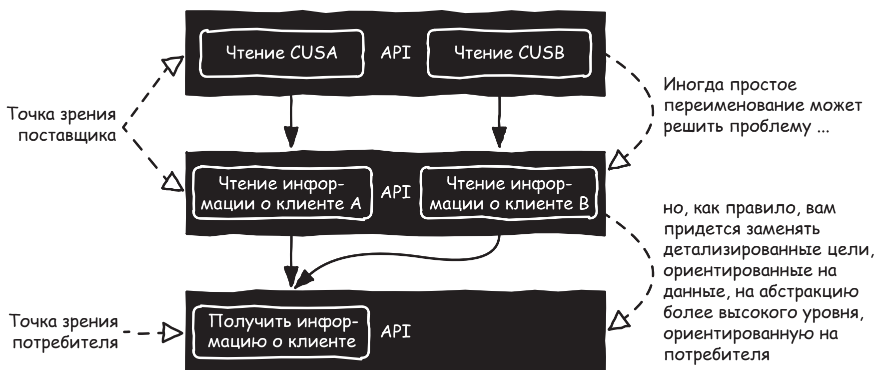



**Рис. 2.18.** Исправляем проблему, связанную с предоставлением доступа к организации данных

Один из вариантов – переименовать цели «чтение CUSA» и «чтение CUSB» в «чтение информации о клиенте A» и «чтение информации о клиенте B», но на самом деле это не улучшает взаимодействие с пользователем. Такие цели, даже если их значение легче понять, по-прежнему отражают точку зрения поставщика. Было бы лучше заменить эти две детализированные, ориентированные на данные цели на одну, более высокоуровневую цель «Получить информацию о клиенте», которая будет более ориентирована на потребителя и удобнее для понимания и использования.

Сопоставление организации данных и имен с целями и данными API может затруднить понимание и использование API. Использование таблицы целей API и сосредоточение внимания на том, что могут делать пользователи, должно позволить вам с легкостью избежать подобных проблем проектирования, но тем не менее они все же могут возникнуть. Поэтому, когда вы определяете цели API, вы всегда должны проверять, не мешаете ли вы потребителям, без надобности открывая доступ к своей модели данных.

Предоставление доступа к модели данных – наиболее очевидный признак точки зрения поставщика, но это не единственный способ, с помощью которого она может проявляться. То, как мы манипулируем данными, также может быть раскрыто через API, и это тоже недопустимо.

### 2.4.2 Как избежать влияния кода и бизнес-логики

Код, управляющий данными – бизнес-логика реализации, – может влиять на проектирование API. Открытие доступа к такой логике через API может докучать не только потребителю, но и поставщику. На рис. 2.19 показан такой пример.

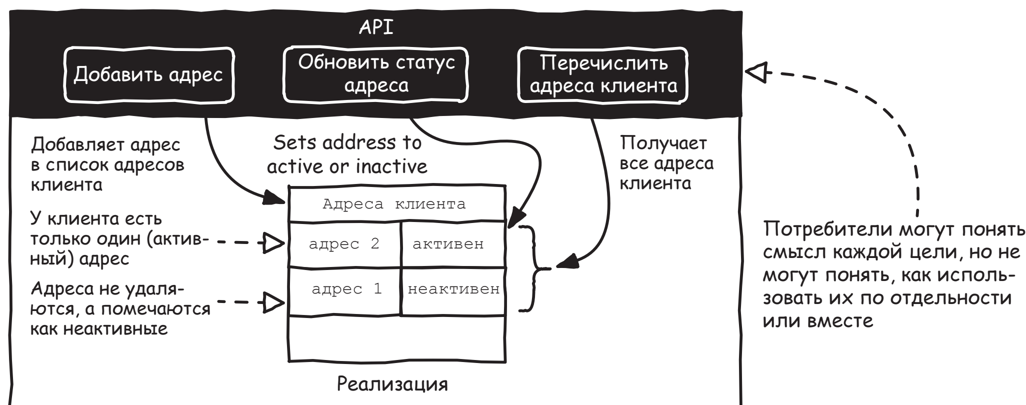



Рис. 2.19. Предоставление доступа к бизнес-логике через API

Допустим, что для реализации нашего API онлайн-магазина у каждого клиента есть один активный адрес. Но адреса в системе никогда не удаляются; вместо этого их *статус* становится *неактивным*, когда клиент не пользуется услугами сайта. Проектирование API, на который вли-

яет эта бизнес-логика, может обеспечить следующие ориентированные на поставщика цели:

- составляем список адресов клиентов (активных и неактивных);
- добавляем адрес клиента;
- обновляем статус адреса (на активный или неактивный).

Слова, используемые для описания этих целей, понятны, но общее назначение целей может быть неочевидно для потребителя, который не знает точно, как система обрабатывает адреса. Эти цели показывают, как данные обрабатываются внутри; на рис. 2.20 показано, как их нужно использовать.

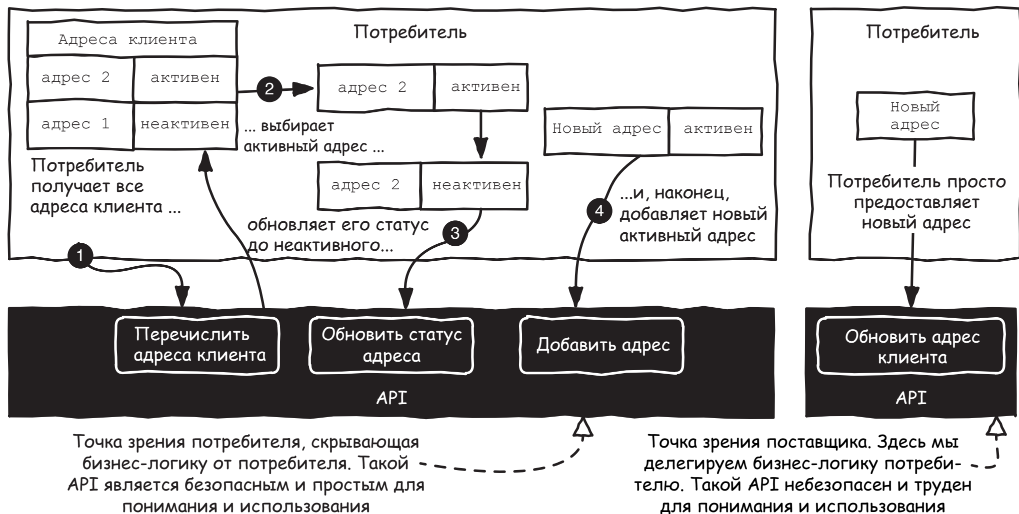



**Рис. 2.20.** Исправляем проблему, связанную с предоставлением доступа к бизнес-логике

В левой части рисунка показано, как изменить адрес клиента с помощью этого API. Потребители должны перечислить существующие адреса, чтобы определить активный, обновить их, чтобы установить статус «неактивный», а затем добавить новый активный адрес. Довольно просто, нет? Вовсе нет! Это очень сложный процесс, и здесь легко что-нибудь может пойти не так. Что, если потребители не установят для предыдущего адреса значение «неактивный»? Это может стать катастрофой для целостности данных – здесь рискует поставщик.

К счастью, в правой части рисунка показано простое решение. Весь этот сложный и опасный беспорядок можно заменить простой целью «Обновить адрес клиента». Реализация может сделать все остальное, как если бы мы позволили схеме обрабатывать цикл включения/выключения магнетрона для API-интерфейса нашего кухонного радара 3000.

Открытие доступа к внутренней бизнес-логике может затруднить использование и понимание API-интерфейса для потребителя и стать опасным для поставщика. Опять же, если вы будете использовать таблицу целей API и сосредоточитесь на том, что могут делать пользователи,

это должно позволить вам с легкостью избежать подобных проблем разработки, но тем не менее они все же могут происходить.

**ПОДСКАЗКА.** При определении целей API всегда следует проверять, чтобы вы случайно не предоставили доступ к внутренней бизнес-логике, которая не касается потребителя и которая может быть опасной для поставщика.

Здесь мы имели дело только с одним программным компонентом. Как может произойти, если мы хотим создавать API на базе более сложной системы, включающей в себя несколько приложений, взаимодействующих друг с другом? Точка зрения поставщика также здесь является проблемой.

### *2.4.3 Как избежать влияния архитектуры программного обеспечения*

Отчасти благодаря API очень часто создаются системы на базе разных частей программного обеспечения, которые взаимодействуют друг с другом (вспомните раздел 1.1.2 в главе 1).

Такая программная архитектура может влиять на проектирование API так же, как и внутренняя бизнес-логика. На рис. 2.21 показан такой пример.

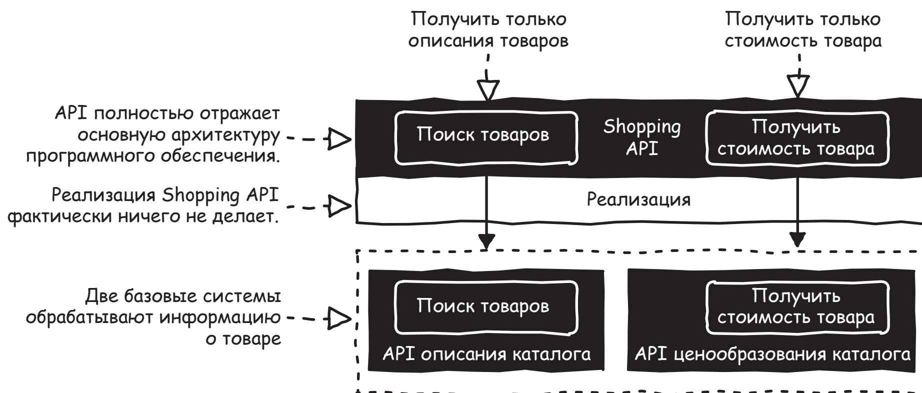



**Рис. 2.21.** Представление доступа к архитектуре программного обеспечения через API

Допустим, что для нашей системы онлайн-покупок мы решили обрабатывать описания товаров и цены на них в двух разных серверных приложениях. На это есть много хороших (и плохих) причин, но здесь дело не в этом. Каковы бы ни были причины, это означает, что информация о товаре хранится в двух разных системах: Описании каталога и Ценообразовании каталога. API онлайн-магазина, спроектированный с точки зрения поставщика, может прямо предоставить доступ к этим двум системам с помощью цели «Поиск товаров», которая только извлекает описания товаров, и цели «Получить стоимость товара», возвращающей его цену. Что это значит для потребителей? Ничего хорошего. Давайте посмотрим на рис. 2.22.

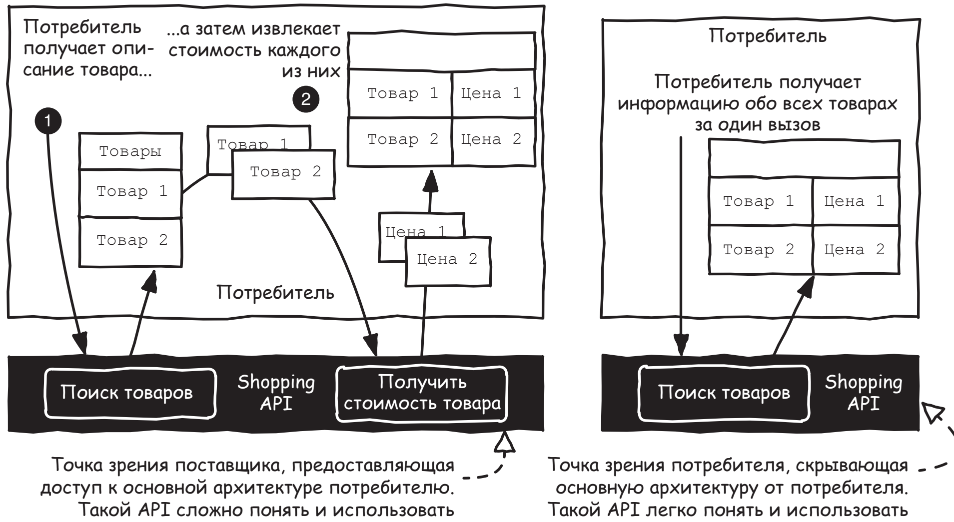



**Рис. 2.22.** **Исправляем проблему, связанную с предоставлением доступа к архитектуре программного обеспечения**

В левой части показано, что для поиска товаров и отображения релевантной информации клиенту (например, описания и цены) потребителю может сначала понадобиться использовать цель «Поиск товаров», чтобы получить описания, а затем получить цену каждого найденного товара, используя цель «Получить стоимость товара», чтобы собрать всю необходимую информацию. Это не очень удобно.

Потребителей не волнует ваш выбор архитектуры программного обеспечения. Им важна вся информация о товарах, которые они ищут. Было бы лучше обеспечить единый поиск товаров и позволить реализации собирать необходимую информацию из базовых систем Описание каталога и Ценообразование каталога, как показано в правой части рисунка.

Сопоставление дизайна API с базовой программной архитектурой может затруднить понимание и использование API потребителями. Повторюсь, если вы будете использовать таблицу целей API и сосредоточитесь на том, что могут делать пользователи, это должно позволить вам с легкостью избежать подобных проблем проектирования, но тем не менее они все же могут возникать. Поэтому, определяя цели API, вы всегда должны проверять, что ваши «что» и «как» не являются результатом вашей базовой архитектуры программного обеспечения.

Мы почти закончили с исследованием различных аспектов точки зрения поставщика. После данных, кода, бизнес-логики и архитектуры программного обеспечения остался только один пункт. И он самый коварный – это организация, где работает много людей.

### 2.4.4 Как избежать влияния организации, где работают люди

Если в организации, предоставляющей API, более одного человека, вы будете сталкиваться с аспектом *человеческой организации* с точки зрения

поставщика. Это основной источник закона Конвея, упомянутого ранее. Люди группируются по отделам или командам. Все эти группы взаимодействуют и общаются по-разному, используя различные процессы, которые неизбежно будут формировать разные системы внутри организации, включая API.

Допустим, наша организация, предоставляющая API-интерфейс онлайн-магазина, разделена на три различных отдела: отдел заказов, который обрабатывает заказы клиентов, отдел складов, занимающийся складом и упаковкой товаров, и отдел отгрузки, имеющий дело с отгрузкой посылок клиентам. На рис. 2.23 показан получившийся в результате API-интерфейс, спроектированный с точки зрения поставщика и потребителя.

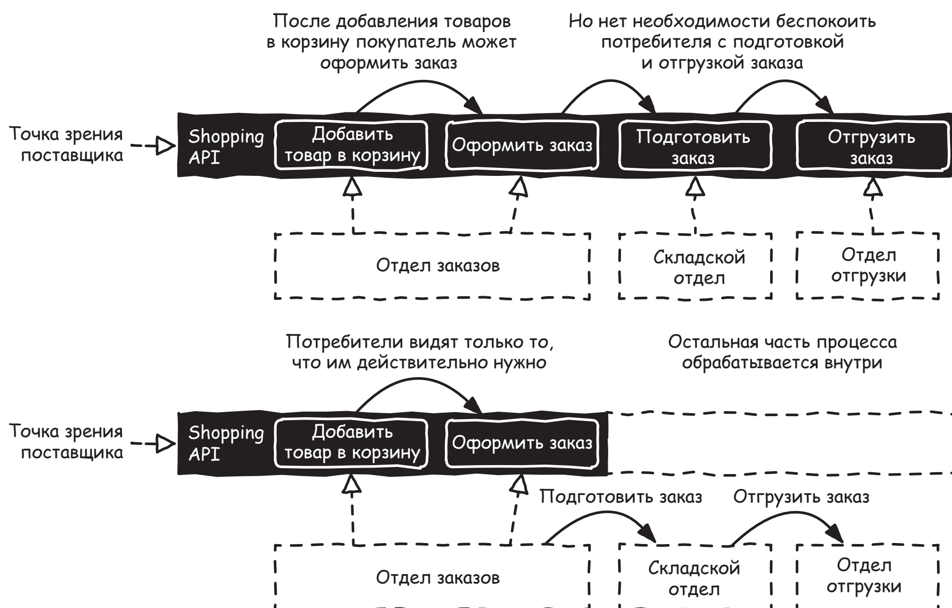



**Рис. 2.23. Как избежать предоставления доступа к действиям такой организации**

При проектировании с точки зрения поставщика наш API может открывать доступ к целям, соответствующим этой организации. Проблема с этими целями состоит в том, что они будут раскрывать внутреннюю работу организации таким образом, что это совершенно не относится к людям, находящимся вне организации.

Если потребители хотят заказать товары для клиентов, им нужно будет использовать цели «Добавить товар в корзину», «Оформить заказ», «Подготовить заказ» и «Отгрузить заказ». Мы снова предоставляем доступ к вещам, которые не касаются потребителя. Какова причина, по которой потребителю нужно использовать цели «Подготовить заказ» и «Доставить заказ»? Ее нет. С точки зрения потребителя, все должно закончиться на цели «Оформить заказ». Когда потребитель использует цель «Оформить заказ», реализация должна работать с отделом складов, чтобы инициировать подготовку заказа, а отдел складов должен иметь дело с отделом от-

грузки, чтобы инициировать доставку заказа. Оставшаяся часть процесса должна обрабатываться внутри.

Сопоставление дизайна API с подобного рода организацией может сделать API непростым для понимания, сложным в использовании и даже совершенно неуместным. Если вы будете использовать таблицу целей API и сосредоточитесь на том, что могут делать пользователи, это должно позволить вам с легкостью избежать подобных проблем разработки, но тем не менее они все же могут происходить. Поэтому, когда вы определяете цели API, всегда следует проверять, действительно ли ваши «что» и «как» имеют отношение к потребителю.

В конце концов, различные аспекты точки зрения поставщика связаны с предоставлением доступа к аспектам, которые не касаются потребителя, через API. Давайте посмотрим, как включить это в нашу таблицу целей API.

### 2.4.5 Определение точки зрения поставщика в таблице целей API

Вы узнали, что для определения целей API вам необходимо ответить на вопросы «кто?», «что?» и «как?», выяснить каковы исходные данные и их источники, и, наконец, разобраться с результатами и их использованием. Используя этот метод, вы можете избежать наиболее очевидных вторжений точки зрения поставщика, не слишком задумываясь об этом. Но вы также убедились, что точка зрения поставщика может быть коварной и не очень очевидной. К счастью, наши исследования точки зрения поставщика показали нам различные аспекты и то, как ее можно обнаружить, просто подумав, действительно ли то, что мы определили, касается потребителя. На рис. 2.24 показан окончательный и обновленный вариант нашей таблицы целей API, чтобы мы были полностью готовы к созданию полного и ориентированного на потребителя списка целей API.

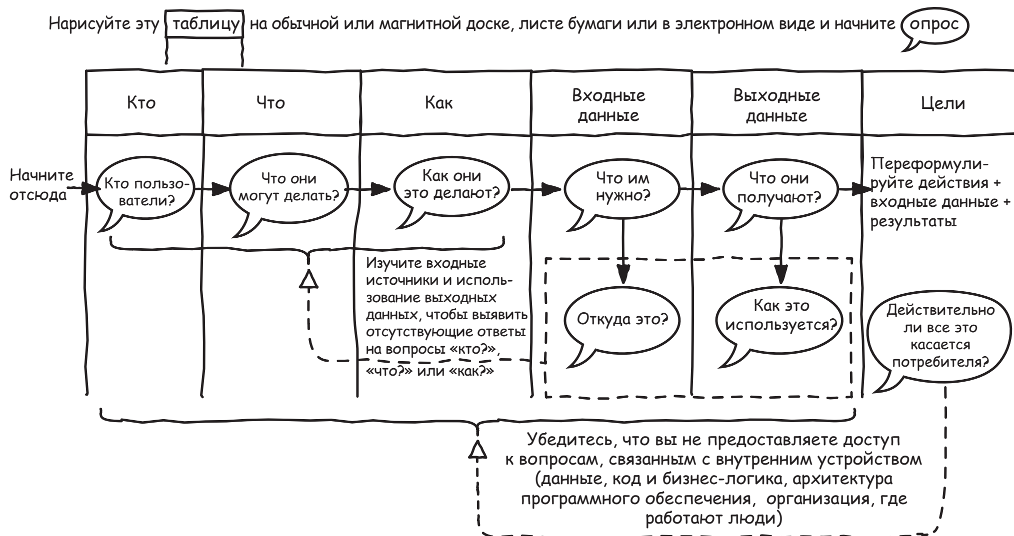



**Рис. 2.24. Обновленная таблица API-целей**

Как видно, все, что нам нужно сделать, – это добавить последний вопрос: «Все это на самом деле касается потребителя?» С помощью этого вопроса, как вы уже видели, мы проверим, берет ли какой-либо элемент свое начало из точки зрения поставщика (данные, код и бизнес-логика, архитектура программного обеспечения или человеческая организация). Список целей API, созданный с помощью этой таблицы, станет прочной основой для проектирования программного интерфейса. В следующей главе вы узнаете, как спроектировать такой интерфейс, основываясь на списке целей API.

## Резюме

- Чтобы потребителям было легко понять и использовать API, он должен проектироваться с точки зрения потребителя.
- Проектирование API с точки зрения поставщика путем прямого предоставления доступа к внутреннему устройству (данные, код и бизнес-логика, архитектура программного обеспечения и человеческая организация) неизбежно приводит к появлению сложных для понимания и сложных в использовании API.
- Комплексный и ориентированный на потребителя список целей API является самой прочной основой для API.
- Определение пользователей, того, что они могут делать, того, как они это делают, что им нужно для этого и что они получают взамен, является ключом к созданию исчерпывающего списка целей API.

3

# Проектирование программного интерфейса

**В этой главе мы рассмотрим:**

- перенос целей API в программный интерфейс;
- определение и отображение ресурсов и действий REST ;
- проектирование данных API из концепций;
- различия между REST API и архитектурным стилем REST;
- почему архитектурный стиль REST важен для проектирования API.

В предыдущей главе вы узнали, как определить цели API; это то, чего пользователи могут достичь, используя его. В случае с API онлайн-магазина некоторые из этих целей могут заключаться в поиске товаров, получении товара, добавлении товара в корзину, оформлении заказа или составлении списка заказов. Эти цели формируют функциональный план API, который мы будем использовать для проектирования фактического программного интерфейса, применяемый его разработчиками и их программным обеспечением. Чтобы спроектировать этот программный интерфейс, мы переносить эти цели и их входные и выходные данные в соответствии со стилем API, как показано на рис. 3.1.

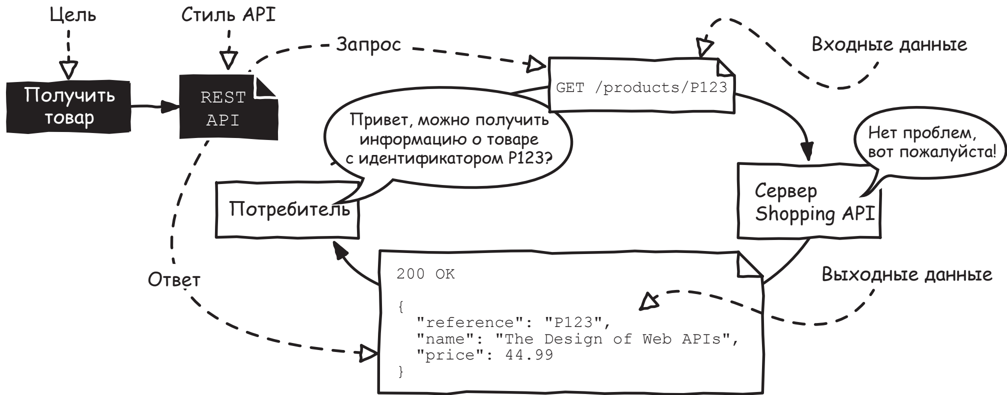



**Рис. 3.1.** Программный интерфейс REST для цели «Получить товар»

REST расшифровывается как *Representational State Transfer* – «передача состояния представления». Здесь REST API переносит цель «Получить товар» в программный интерфейс. Он представлен запросом `GET /products/{productId}`, где productId - это исходный параметр (здесь его значение равно `P123`), а `200 OK` – это ответ с выходными данными, состоящими из свойства `reference`, `name` и `price`. Как спроектировать такой программный интерфейс?

Представление целей с использованием REST или любого другого типа программного интерфейса требует, чтобы вы для начала понимали, как он работает. Действительно, что означают, например, эти `GET /products/{productId}` или `200 OK`? Не зная этого, вы не сможете разработать такой интерфейс. Если у вас есть базовые знания, можете анализировать цели и представлять их в соответствии с выбранным стилем API. Вам также нужно более точно проектировать данные, которыми обмениваются через API, по сравнению с тем, что мы делали при заполнении таблицы целей API. Этот процесс похож на то, что вы делаете, когда программируете.

Звучит довольно просто, не правда ли? Но это не так. Изучив, как переносить основные цели в программный интерфейс, вы, возможно, поймете, что некоторые из ваших целей трудно представить. В таких случаях нужно найти путь между удобством для пользователя и соблюдением выбранного стиля API, чтобы придумать лучшее из возможных представлений.

После всего этого вы можете задаться вопросом, что на самом деле означает «передача состояния представления» и почему она была выбрана в качестве основного примера программного интерфейса для этой книги. Зачем использовать REST API, чтобы научить проектированию API? Чем эти API лучше других? Несмотря на то что они получили широкое распространение, за этим выбором стоит гораздо более важная причина: REST API основаны на архитектурном стиле REST, опирающемся на прочную основу, которую полезно знать при проектировании любого API-интерфейса. Мы скоро к этому вернемся, но обо всем по порядку. Давайте поговорим о базовых принципах REST API.

## 3.1 Знакомство с REST API

Чтобы получить достаточное представление о REST API, для того чтобы спроектировать один из таких интерфейсов, мы проанализируем вызов REST API, сделанный потребителем API-интерфейса онлайн-магазина для получения информации о товаре, как показано во введении к этой главе (рис. 3.1). Мы будем считать само собой разумеющимся, что REST-представление этой цели – `GET /products/{productId}`, и будем работать с примером `GET /products/P123`. Если вы помните раздел 1.1.1, то должны догадаться, что этот запрос как-то связан с протоколом HTTP. Данный анализ покажет нам, что HTTP фактически используется этим вызовом. После этого мы сможем подробно изучить протокол HTTP и базовые принципы REST API.

### 3.1.1 Анализ вызова REST API

Что происходит, когда потребитель хочет достичь цели «Получить товар»? Или, говоря более конкретно, что происходит, когда он хочет получить подробную информацию о товаре с идентификатором `P123` из каталога товаров с помощью REST API онлайн-магазина? Потребители должны обмениваться данными с сервером, на котором размещен API, используя протокол HTTP, как показано на рис. 3.2.

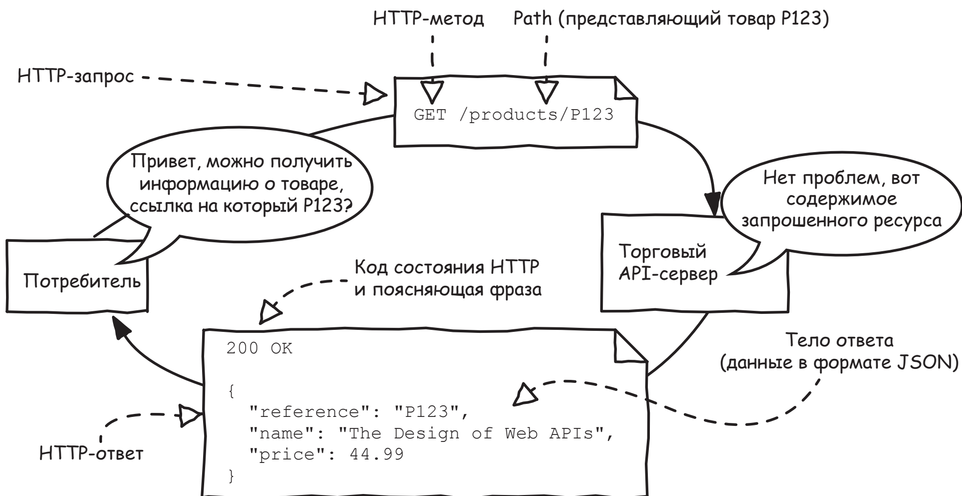



**Рис. 3.2.** Вызов REST API с использованием протокола HTTP

Поскольку эта цель представлена `GET /products/{productId}`, потребитель должен отправить HTTP-запрос `GET /products/P123` на сервер API онлайн-магазина. В ответ сервер возвращает ответ, `200 OK`, за которым следует информация о запрашиваемом товаре. (Обратите внимание, что мы упростили этот обмен данными по протоколу HTTP, чтобы сосредоточиться только на тех элементах, которые важны для нас.)

Запрос состоит из HTTP-метода `GET` и пути `/products/P123`. *Путь* – это адрес, идентифицирующий ресурс на сервере; в данном случае товар

P123 находится в products. HTTP-*метод* указывает на то, что потребитель хочет сделать с этим ресурсом: GET означает, что он хочет получить ресурс. С функциональной точки зрения такой запрос означает что-то вроде: «Привет, могу ли я получить информацию о товаре с идентификатором P123?» Но с точки зрения протокола HTTP это означает: «Привет, могу ли я определить ресурс по пути /products/P123?»

Первая часть ответа состоит из кода состояния HTTP 200 и поясняющей фразы OK. *Код состояния* сообщает нам, как прошла обработка запроса. Благодаря поясняющей фразе мы можем догадаться, что код состояния HTTP 200 означает, что все прошло нормально. Вторая часть ответа называется *телом ответа*. Она содержит содержимое ресурса, идентифицированного путем в запросе, который в данном случае представляет собой информацию о товаре P123 в виде данных в формате JSON.

С функциональной точки зрения ответ, возвращаемый сервером API, в основном означает: «Конечно, вот информация о запрашиваемом товаре». С точки зрения HTTP имеется в виду: «Нет проблем, вот содержимое запрошенного ресурса».

## Формат данных JSON

JSON – текстовый формат данных, основанный на том, как язык программирования JavaScript описывает данные, но, несмотря на свое название, полностью не зависим от языка (см. https://www.json.org/). Используя JSON, вы можете описывать объекты, содержащие неупорядоченные пары типа имя/значение, а также массивы или списки, содержащие упорядоченные значения, как показано на этом рисунке.

```
{
  "aString": "a string value",
  "aNumber": 1.23,
  "aBoolean": true,
  "aNullValue": null
  "anObject": {
    "name": "value"
  }
}
```

Объект JSON

```
[
  { "aString": "one",
    "aNumber": 1 },
  { "aString": "two",
    "aNumber": 2 },
  { "aString": "three",
    "aNumber": 3 }
]
```

Массив JSON

**Пример документов в формате JSON**

Объект ограничен фигурными скобками ({}). Имя – это строка в кавычках ("name"), отделенная от значения двоеточием (:). Значением может быть строка типа "value", число типа 1.23, логическое значение (true или false), нулевое значение null, объект или массив. Массив ограничивается скобками ([]), а его значения разделены запятыми (,).

Формат JSON легко преобразуется с помощью любого языка программирования. Его также относительно легко читать и писать. Он широко применяется для многих целей, таких как базы данных, конфигурационные файлы и, конечно же, API.

Теперь вы знаете, как потребители могут вызывать API онлайн-магазина, чтобы получить товар. Но протокол HTTP создан не только для получения документов в формате JSON.

### 3.1.2 Базовые принципы HTTP

HTTP является основой обмена данными для Всемирной паутины. Это независимый от языка программирования протокол, предназначенный для обмена документами (также называемыми *ресурсами*) между приложениями через интернет. Протокол HTTP используется широким спектром приложений, наиболее известными из которых являются веб-браузеры.

Веб-браузер использует протокол HTTP для связи с веб-сервером, на котором размещен сайт. Когда вы набираете URL-адрес (например, http://apihandyman.io/about) в адресной строке браузера, он отправляет HTTP-запрос `GET /about` на сервер, где находится `apihandyman.io`, так же как когда потребитель API отправляет запрос на сервер REST API. Ответ, отправленный сервером, содержит код состояния `200 OK`, за которым следует HTML-страница, соответствующая URL-адресу.

Браузеры используют этот протокол для извлечения любого типа ресурса (документа): HTML-страниц, CSS-файлов, файлов JavaScript, изображений и любых других документов, которые необходимы веб-сайту. Но это не единственное его использование. Когда вы, например, загружаете фотографию на сайт социальной сети, браузер использует протокол HTTP, но на этот раз для отправки документа на сервер. В этом случае браузер отправляет запрос `POST /photos` с телом, содержащим файл изображения. Поэтому протокол HTTP также может использоваться для отправки содержимого ресурса.

HTTP-запросы и ответы всегда выглядят одинаково независимо от того, что запрашивается и каков результат обработки запроса (рис. 3.3).

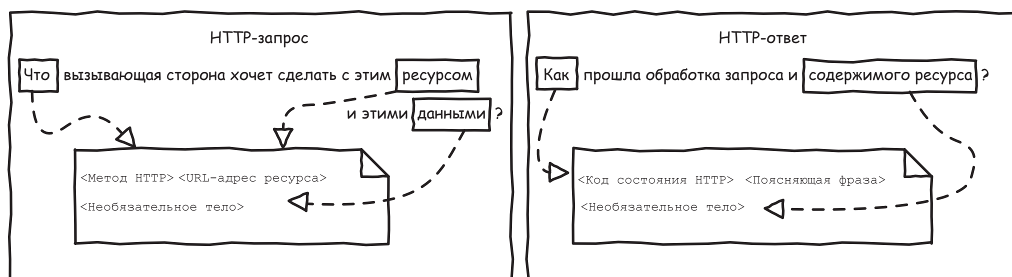



**Рис. 3.3. Базовая структура HTTP-запроса и ответа**

Каким бы ни было его назначение, базовый HTTP-запрос содержит метод HTTP и путь к ресурсу. *Метод* HTTP указывает, что нужно сделать с ресурсом, который идентифицирует путь. Вы уже видели два HTTP-метода – `GET`, используемый для получения ресурса, и `POST`, используемый для его отправки, – позже вы узнаете больше.

За этой первой частью запроса может следовать тело с содержимым ресурса, которое необходимо отправить на сервер, например для созда-

ния, обновления или замены ресурса. Это содержимое может быть любого типа: документ в формате JSON, текстовый файл или фотография.

Как упоминалось ранее, ответ HTTP, возвращаемый сервером, всегда содержит код состояния и поясняющую фразу. Это указывает на то, как прошла обработка запроса – была ли она успешной или нет. Пока вы видели только один код состояния HTTP, `200 OK`, но позже познакомитесь с другими кодами (например, с известным кодом `404 NOT FOUND`, о чем пойдет речь в разделе 5.2.3). За этой первой частью ответа может следовать тело с содержимым ресурса, которым манипулировал запрос.

Как и тело запроса, тип этого содержимого может быть любым.

Протокол HTTP кажется довольно простым. Но настолько ли просты REST API, которые используют его?

### 3.1.3 Базовые принципы REST API

Вы убедились, что, используя REST API онлайн-магазина, чтобы получить товар, потребители должны отправить HTTP-запрос на сервер, где размещен API. Этот запрос использует HTTP-метод GET для пути `/products/{productId}`, который идентифицирует товар. Если все в порядке, сервер возвращает ответ, содержащий HTTP-статус, указывающий на это вместе с данными о товаре. Вы также видели, что, если веб-браузер хочет получить страницу из моего блога `apihandyman.io`, он отправляет HTTP-запрос. Этот запрос использует HTTP-метод `GET` для пути к странице (например, `/about`). Если все в порядке, веб-сервер также возвращает ответ, содержащий код `200 OK` и содержимое страницы. Происходит все то же самое!

И веб-сервер, и сервер с API онлайн-магазина предоставляют доступ к HTTP-интерфейсу, который учитывает ожидаемое поведение протокола HTTP. Базовый REST API не только использует протокол HTTP – он полностью опирается на него, как показано на рис. 3.4.

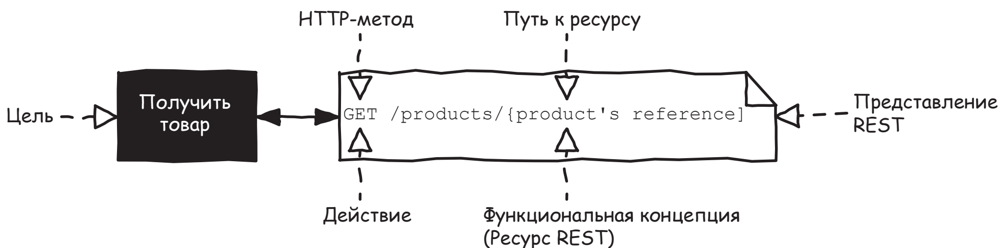



**Рис. 3.4. Отображение цели в HTTP-запрос**

Чтобы позволить своим потребителям достигать своих целей, REST API дает им возможность манипулировать ресурсами, которые идентифицируются путями, с помощью стандартизированных методов HTTP. *Ресурс* – это функциональная концепция. Например, `/products/{productId}` распознает конкретный товар в каталоге товаров. Этот путь идентифицирует ресурс товара. HTTP-метод `GET` представляет действие получения, которое можно применить к этому ресурсу для фактического получения товара.

Вызов REST API – не что иное, как HTTP-вызов. Давайте теперь посмотрим, как перейти от цели «Получить товар» к `GET /products/{product's reference}` — как преобразовать цели в HTTP-метод и пары путей.

## 3.2 Перенос целей в REST API

Вы узнали, что REST API представляет свои цели, используя протокол HTTP.

Цели переносятся в пары типа «ресурс и действие». Ресурсы идентифицируются путями, а действия представлены методами HTTP. Но как идентифицировать эти ресурсы и действия? И как обозначать их, используя пути и методы HTTP?

Мы делаем то, что всегда делалось при проектировании программного обеспечения. Мы анализируем наши функциональные потребности для определения ресурсов и того, что с ними происходит, прежде чем переносить их в программное представление. Существует множество методов проектирования программного обеспечения, которые можно использовать для идентификации ресурсов и того, что можно с ними делать, основываясь на таких спецификациях, как таблица API-целей из предыдущей главы. Однако в этой книге показан очень простой метод, состоящий из четырех этапов (см. рис. 3.5).

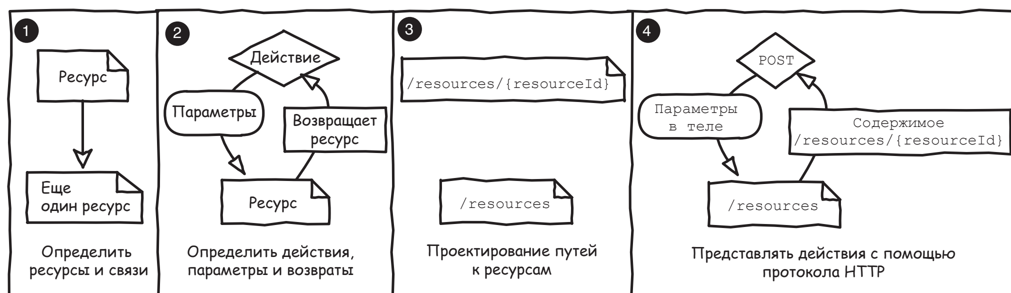



**Рис. 3.5. От целей к REST API**

Сначала мы должны идентифицировать ресурсы (функциональные концепции) и их связь (как они организованы). Затем нужно определить доступные действия для каждого ресурса, их параметры и результаты. Как только это будет сделано, мы сможем перейти к фактическому проектированию программного интерфейса, создав пути к ресурсам и выбрав HTTP-методы для обозначения действий.

В последующих разделах рассмотрим этот процесс более подробно. Здесь речь идет только о номинальном случае, когда все в порядке – `200 OK`. Мы поговорим об обработке ошибок в разделе 5.2.

> **ПРИМЕЧАНИЕ.** Как только вы ознакомитесь с процессом отображения целей в пути к ресурсам и HTTP-методами, используя этот метод, можете смело адаптировать его или использовать предпочтительный метод проектирования программного обеспечения, если вы достигаете того же результата.

### 3.2.1 Идентификация ресурсов и их связей с таблицей целей API

Таблица целей API, с которой вы познакомились в главе 2, описывает, кто такие пользователи, что они могут делать, как они это делают, что им нужно для этого и что они могут получить взамен. Мы можем использовать эту информацию для определения целей API, которые мы перенесем в REST API. Чтобы попрактиковаться на простом, но полном примере, я усовершенствовал часть «Управление каталогом» в таблице целей API онлайн-магазина, с которым мы начали работать в главе 2 (рис. 3.6).

Эта таблица целей API (см. Главу 2)

Мы перенесем эти цели в REST API

| Кто | Что | Как | Входные данные (источник) | Результаты (использование) | Цели |
|---|---|---|---|---|---|
| Пользователи с правами администратора | Управление каталогом | Добавить товар | Каталог (API), информация о товаре (пользователь) | Добавленный товар (получить, обновить, удалить, заменить) | Добавить товар в каталог |
| | | Получить информацию о товаре | Товар (поиск, добавление) | Информация о товаре (пользователь) | Получить товар |
| | | Обновить информацию о товаре | Товар (получить, найти, добавить), обновленная информация (пользователь) | | Обновить товар |
| | | Заменить товар | Товар (получить, найти, добавить), информация о новом товаре (пользователь) | | Заменить товар |
| | | Удалить товар | Товар (получить, найти, добавить) | | Удалить товар |
| | | Поиск товаров | Каталог (API), бесплатный запрос (пользователь) | Товары, соответствующие запросу (получить, обновить, удалить, заменить) | Поиск товаров в каталоге с помощью бесплатного запроса |

**Рис. 3.6.** Таблица целей API

Как видно на этом рисунке, при управлении каталогом товаров пользователи с правами администратора могут добавлять товар в каталог. Они также могут получать информацию о товаре, обновлять, заменять или удалять его. Наконец, можно искать товары, используя бесплатный запрос. На рис. 3.7 показано, что мы просто анализируем цели API и перечисляем все существительные, к которым применяются основные глаголы целей для идентификации ресурсов.

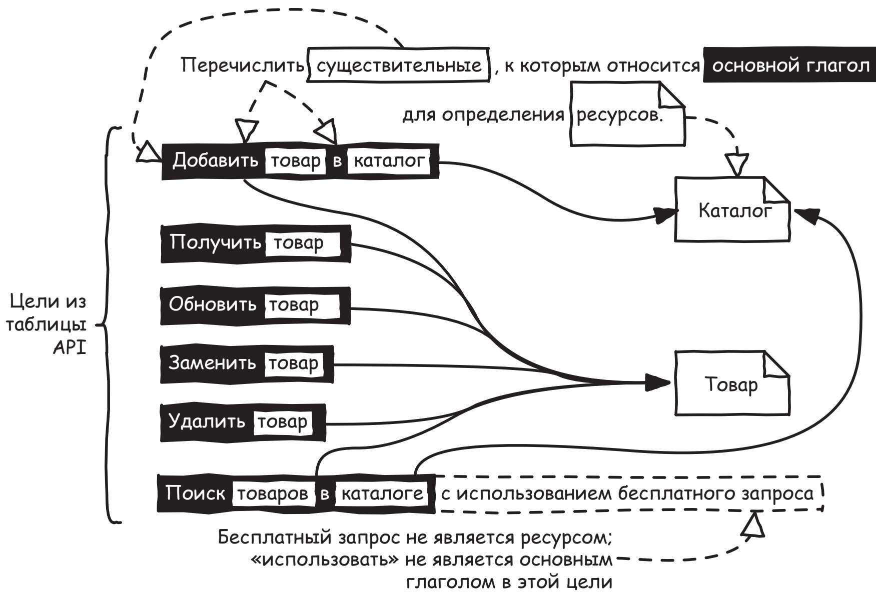



**Рис. 3.7.** Идентификация ресурсов

Когда мы *добавляем* товар в каталог, основным глаголом является *add*. Он применяется как к товару, так и к ресурсам каталога. Когда мы *получаем* товар, основным глаголом будет *get*, применяемый только к товару. Но когда мы *ищем* товары в каталоге, используя бесплатный запрос, *бесплатный запрос* – это существительное, а не ресурс, потому что глагол *search* не применяется к нему напрямую. Таким образом мы можем выделить два ресурса: *каталог* и *товар*.

Теперь давайте посмотрим, как эти ресурсы связаны. Чтобы определить организацию ресурсов, мы перечислим цели, в которых упоминается более одного ресурса (см. рис. 3.8).

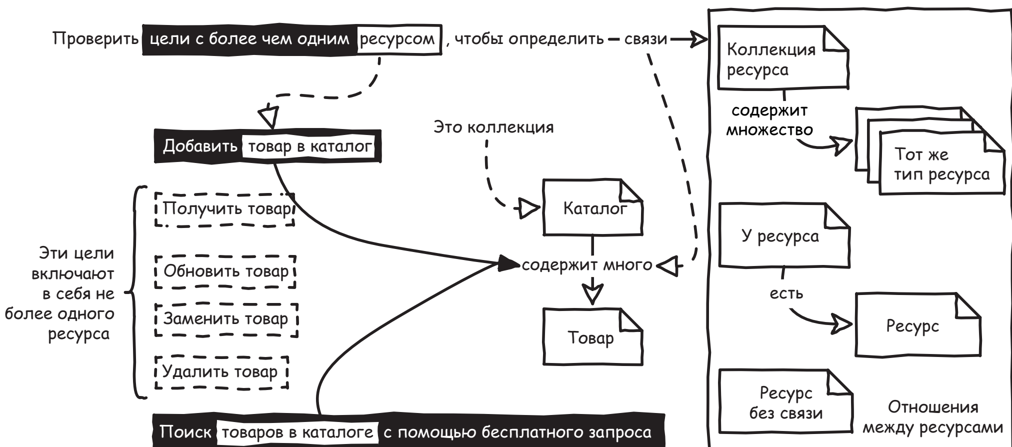



**Рис. 3.8.** Идентификация связей между ресурсами

У нас есть две цели, связанные с более чем одним ресурсом. В первой мы *добавляем* товар в каталог. Во второй мы *ищем* товары в каталоге.

Ресурсы могут иметь или не иметь связи с другими ресурсами, а ресурс может содержать какие-либо другие ресурсы того же типа и называться *ресурсом-коллекцией* или просто *коллекцией*. В нашем случае ресурс каталога является коллекцией: он содержит ресурсы товара. Если бы мы проектировали API, связанный с городским планированием, город мог бы стать ресурсом-коллекцией, содержащим множество ресурсов зданий. Один ресурс также может быть связан с другими; например, ресурс здания – с ресурсом адреса.

Мы определили наш каталог и ресурсы товара и то, как они связаны. Что можно с ними сделать?

### 3.2.2 Идентификация действий, их параметров и результатов с помощью таблицы целей API

REST API представляет свои цели с помощью действий над ресурсами. Чтобы идентифицировать действие, мы берем основной глагол цели и связываем его с ресурсом, к которому он применяется, как показано на рис. 3.9.

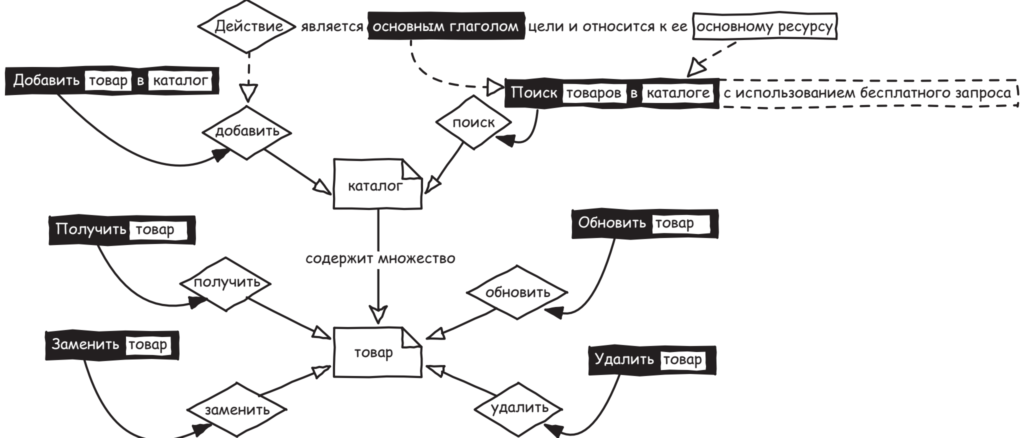



**Рис. 3.9. Идентификация действий**

Что касается целей с одним ресурсом, это просто. Например, «Получить товар», где глагол *get* применяется к ресурсу товара. Но как насчет целей «Добавить товар в каталог» и «Искать товар в каталоге», используя бесплатный запрос? Связываем ли мы добавление и поиск с товаром или каталогом? Мы *добавляем* товар в каталог и *ищем* товары в каталоге; в этом случае, глаголы *add* и *search* связаны с ресурсом каталога. Это означает, что мы связываем глагол с основным ресурсом (каталогом), который используется или изменяется, – другой ресурс (товар) является только параметром или результатом.

Эти действия могут нуждаться в дополнительных параметрах и могут возвращать некую информацию. К счастью, мы уже определили эти

параметры и возвращаемые результаты; таблица целей API поставляется с полным списком входных данных (параметров) и выходных данных (результатов), как показано на рис. 3.10.

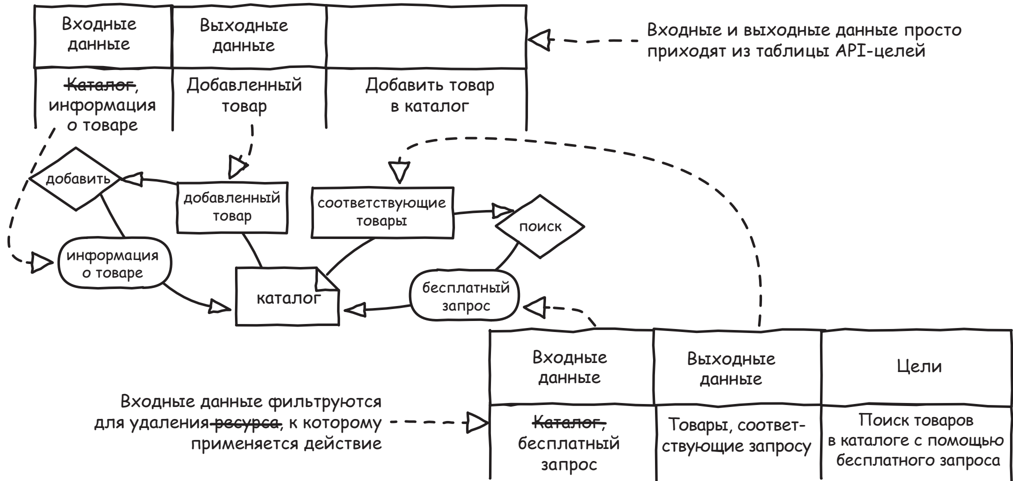



Рис. 3.10. Идентификация параметров действия и результатов

Нам просто нужно отфильтровать входные данные, потому что некоторые из них могут быть ресурсами, к которым применяется действие. Когда мы применяем глагол *add* к ресурсу каталога, нам нужна какая-то информация о товаре в качестве входных данных; и в свою очередь мы получаем недавно созданный ресурс товара. Мы предоставляем бесплатный запрос действию *search*, которое применяется к ресурсу каталога, и оно возвращает соответствующие ресурсы товара. Затем мы делаем то же самое для действий, применяемых к ресурсу товара, – и все готово! Мы идентифицировали все ресурсы и их действия, включая параметры и результаты, проанализировав таблицу API и его цели (рис. 3.11).

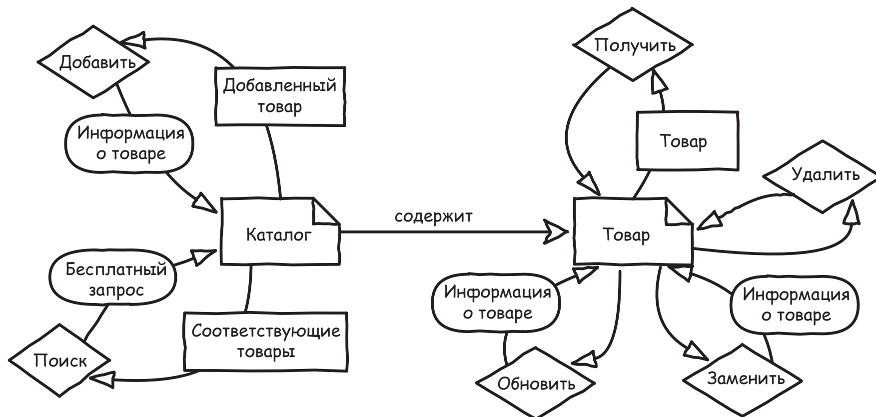



Рис. 3.11. Все идентифицированные ресурсы и действия

Как видно, этот процесс на самом деле не отличается от того, что вы делаете при проектировании реализации программного обеспечения. Чтобы описать в книге, как это сделать, потребуется много времени, но с помощью таблицы API на это уйдет всего пара минут. Давайте теперь посмотрим, как представить все это с помощью протокола HTTP. Начнем с представления ресурсов путями.

### 3.2.3 Представление ресурсов с помощью путей

Анализируя таблицу целей API, мы идентифицировали два ресурса: каталог и товар. Мы также обнаружили, что ресурс каталога представляет собой коллекцию ресурсов товара. Как спроектировать пути этих ресурсов? На рис. 3.12 показано, как это сделать.

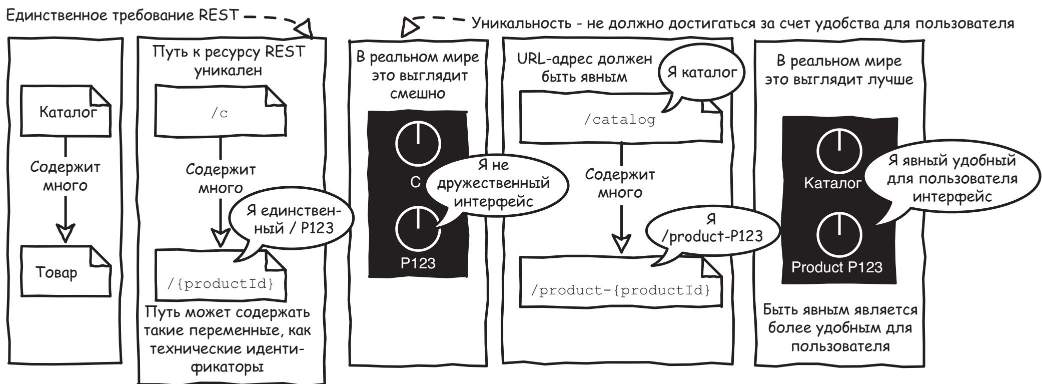



**Рис. 3.12.** Путь ресурса REST. Единственное требование – он должен быть уникальным, но также должен быть явным

Путь к ресурсу REST должен быть уникальным. Чтобы идентифицировать каталог, мы могли бы использовать путь `/c`. В случае с товарами мы могли бы использовать ссылку на товар или технический идентификатор и создать с его помощью путь `/{productId}` (например, `/P123`). Такие переменные в путях называются параметрами пути. Пути `/c` и `/{productId}` являются совершенно допустимыми путями REST, потому что они уникальны.

Но давайте будем откровенны. Что бы вы подумали о таком интерфейсе, если бы увидели его в реальной ситуации? Он не очень удобен для потребителей; и, как вы помните из главы 2, мы всегда должны разрабатывать API для пользователей. Было бы лучше выбрать пути, которые явно указывают, что они обозначают. Почему бы просто не использовать `/catalog` для ресурса каталога и `/product-{productId}` для ресурса товара? Звучит хорошо, но эти пути не единственная возможность, как показано на рис. 3.13.

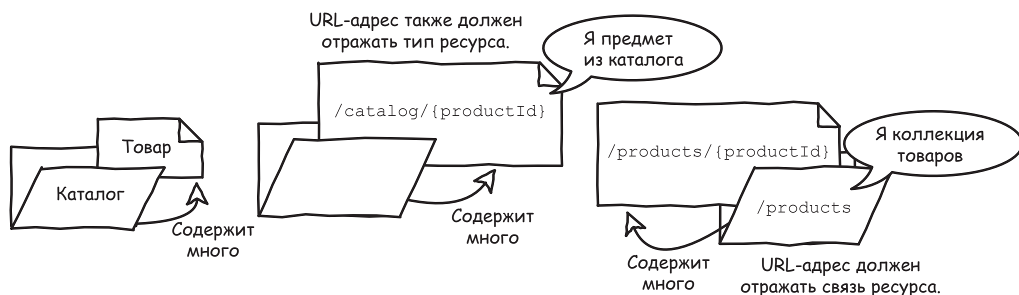



**Рис. 3.13. Путь к ресурсу REST должен показывать иерархию и тип**

Чтобы повысить удобство использования, связь между ресурсами каталога и товаром можно отобразить в путях наподобие иерархии папок в файловой системе. Каждый ресурс товара – это элемент в коллекции каталога, идентифицируемый как /catalog, поэтому мы можем выбрать путь /catalog/{productId} для обозначения товара. Мы также могли бы явно указать, что каталог – это коллекция ресурсов товара, используя путь /products, при этом товар из этой коллекции представлен путем /products/{productId}. Вариантов много! Все они показаны на рис. 3.14.

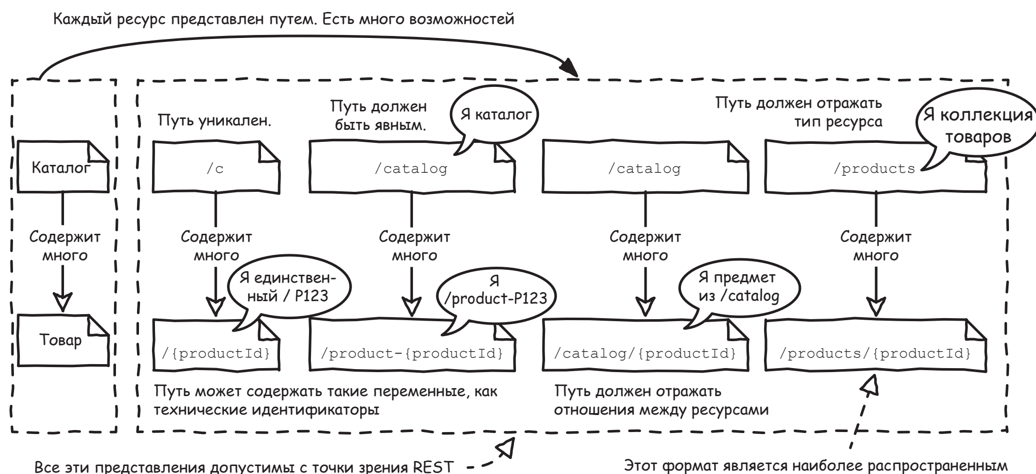



**Рис. 3.14. Выбор формата пути к ресурсу**

С точки зрения чистого REST все эти варианты допустимы. Даже если мы уже отбросили непонятные пути, такие как /c и /{productId}, потому что они явно неудобны для потребителя, у нас все еще есть много возможностей. Каталог ресурса можно представить как /catalog или /products, а ресурс товара как /product-{productId}, /catalog/{productId} или /products/{productId}.

**ПРИМЕЧАНИЕ.** Вы можете выбрать любой вариант, какой предпочитаете, но помните, что пути к ресурсам должны быть удобными для пользователя. Пользователи API должны иметь возможность с лег-

костью расшифровывать их, поэтому чем больше информации вы предоставите в своих путях, тем лучше.

Хотя официальных правил REST, касающихся проектирования путей к ресурсам (кроме уникальности), не существует, наиболее распространенным форматом является `/{имя, отражающее тип элемента коллекции во множественном числе}/{идентификатор элемента}`. Использование путей к ресурсам, показывающих иерархию ресурсов, и использование имен во множественном числе для коллекций для отображения типа элемента коллекции стало де-факто стандартом REST.

Поэтому в нашем примере каталог должен быть идентифицирован как `/products`, а товар – как `/products/ {productId}`. Эту структуру можно расширить до нескольких уровней, например: `/resources/{resourceId}/sub-resources/{subResourceId}`.

Мы почти закончили! Мы определили ресурсы и их действия и разработали пути к ресурсам. Вот последний шаг, представляющий действия с протоколом HTTP.

### 3.2.4 Представление действий с помощью протокола HTTP

Начнем с ресурса каталога и его действия *add*, как показано на рис. 3.15.

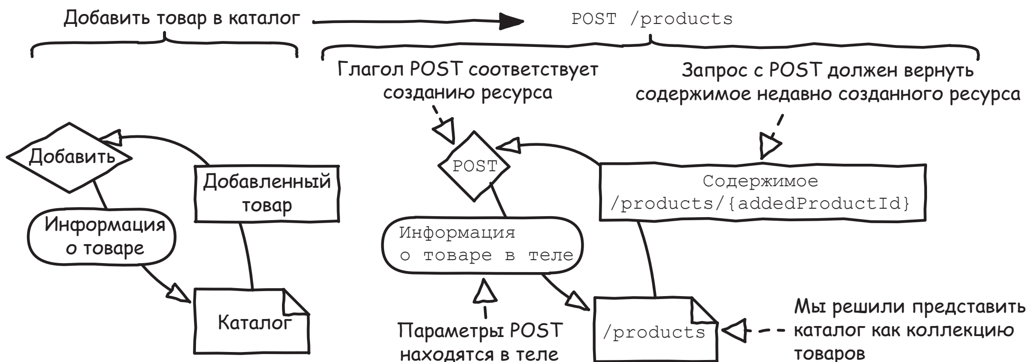



Рис. 3.15. Добавление товара в каталог в виде запроса по протоколу HTTP

HTTP-представление цели «Добавить товар в каталог» – `POST /products`. Когда мы добавляем товар в ресурс каталога, обозначенный как `/products`, мы фактически *создаем* ресурс товара, используя предоставленную информацию о товаре. HTTP-метод, соответствующий созданию ресурса, – это `POST`. Параметры метода запроса `POST` обычно передаются в теле запроса, поэтому параметр информации о товаре отправляется туда. Как только ресурс товара создан, действие должно вернуть вновь созданный ресурс, идентифицируемый по его пути: `/products/{AddedProductId}`.

Теперь как выглядит HTTP-представление действия ресурса каталога *search*?

HTTP-представление поиска товаров в каталоге с использованием бесплатного запроса – `GET /products?free-query={free query}`, как показано на рис. 3.16.

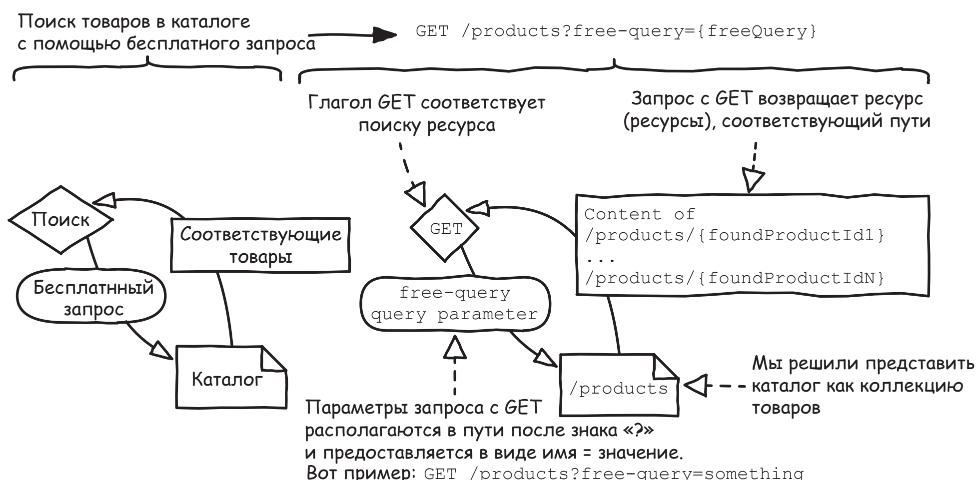



**Рис. 3.16. Поиск товаров в каталоге с использованием бесплатного запроса в качестве HTTP-запроса**

Когда мы ищем товары, мы хотим получить их, поэтому должны использовать HTTP-метод `GET` в пути `/products`. Чтобы получить только те товары, которые соответствуют некоему запросу, например названию товара или частичному описанию, нужно передать параметр в этот запрос.

Параметры HTTP-запроса `GET` обычно предоставляются в виде параметров запроса в пути, как показано в приведенном ниже листинге.

**Листинг 3.1. Примеры параметров запроса**

```
GET /example?param1=value1&param2=value2
GET /products?free-query=something
```

Параметры расположены после знака ? в конце пути и предоставляются в формате имя = значение (например, `param1=value1`). Несколько параметров запроса разделяются символом &. Как только поиск будет завершен, метод запроса `GET` возвращает ресурсы, соответствующие пути (что включает в себя параметр free-query).

Мы представили все действия ресурса каталога, поэтому давайте поработаем с ресурсом товара. Мы начнем с действия «Получить товар», который относительно легко представить в виде HTTP-запроса, как показано на рис. 3.17.

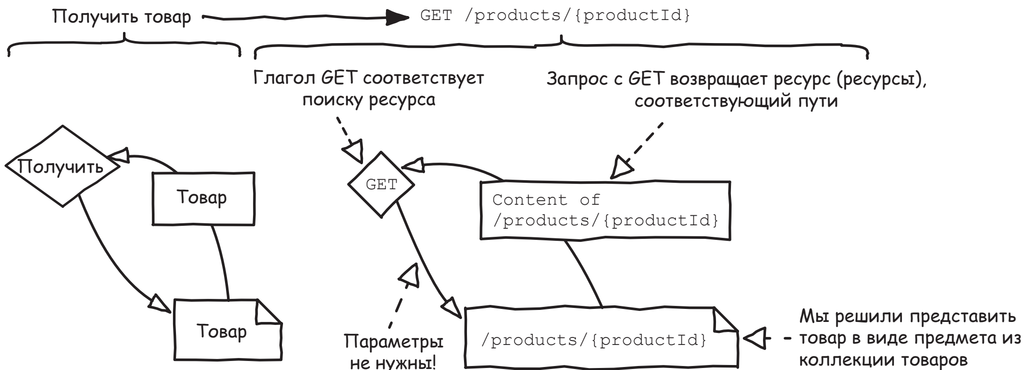



Рис. 3.17. Получение товара в виде HTTP-запроса

Мы хотим получить ресурс товара, обозначенный как /products/{productId}, поэтому снова используем HTTP-метод GET. Следовательно, HTTP-представление – это GET/products/{productId}. GET-ресурс всегда возвращает ресурс, соответствующий предоставленному пути, поэтому это действие возвращает содержимое этого ресурса.

Теперь пришло время познакомиться с новыми методами HTTP! Как представить удаление товара с помощью протокола HTTP? Просто – DELETE /products/{productId}, как показано на рис. 3.18.

Рис. 3.18. Удаление товара в виде HTTP-запроса

Очевидно, что целью HTTP-метода DELETE является удаление ресурса, соответствующего указанному пути. В нашем случае это действие не возвращает никакой информации.

Удалить товар было легко. А теперь вы можете догадаться, какой HTTP-метод мы будем использовать для обновления товара? Тут есть одна хитрость: HTTP-представление обновления товара – это PATCH /products/{productId}, а не UPDATE /products/{productId}, как показано на рис. 3.19.

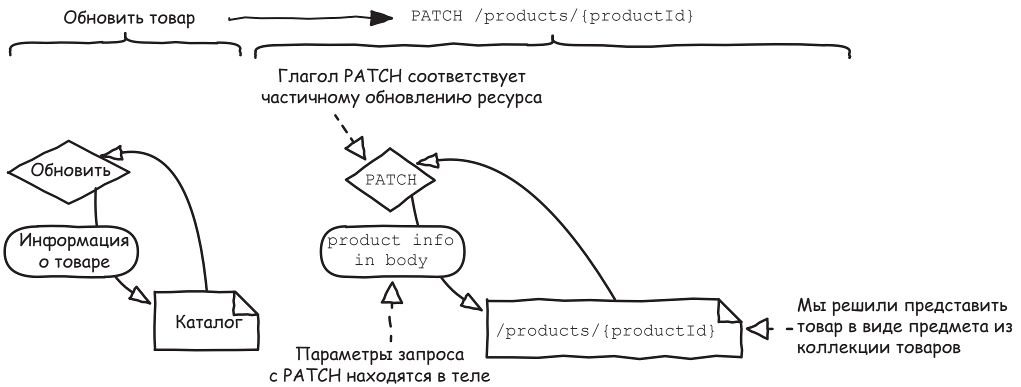



**Рис. 3.19. Обновление товара в виде HTTP-запроса**

HTTP-метод `PATCH` можно использовать для частичного обновления ресурса. Как и в случае с `POST`, параметры запроса передаются в теле запроса. Например, если вы хотите обновить цену товара – можно использовать метод `PATCH` для ресурса товара и передать обновленную цену в теле. В нашем случае использования это действие не возвращает никакой информации.

В нашем последнем примере показан HTTP-метод, имеющий две цели. HTTP-представление замены товара – `PUT /products/{productId}`, как показано на рис. 3.20.

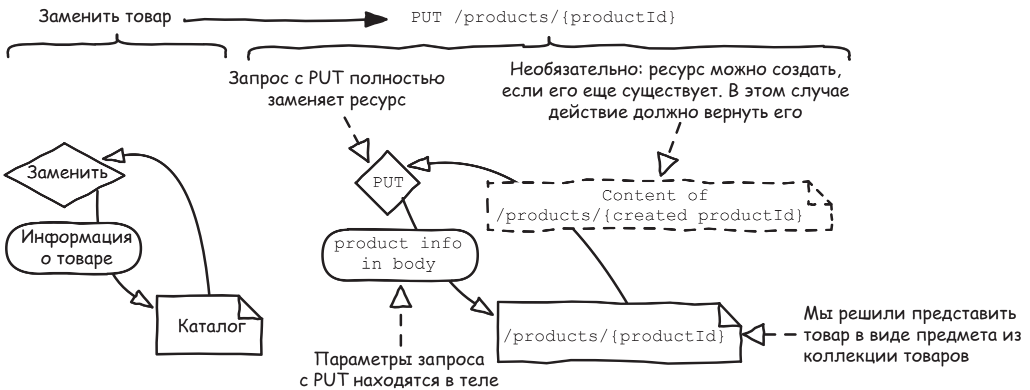



**Рис. 3.20. Замена товара в виде HTTP-запроса**

HTTP-метод `PUT` можно использовать для полной замены существующего ресурса или для создания несуществующего и предоставления его идентификатора. В последнем случае эффект будет тот же, что и при добавлении товара в каталог. Как и в случае с `POST`, параметры запроса передаются в теле запроса. В нашем случае использования это действие не возвращает информацию, но, если вы используете метод `PUT` для создания ресурса, созданный ресурс должен быть возвращен.

Таким образом, HTTP-методы `POST`, `GET`, `PUT`, `PATCH` и `DELETE`, по сути, отображают базовые функции CRUD (создание, чтение, обновление, удаление). Не забывайте, что эти действия выполняются с точки зрения потребителя; например, если вы выполните действие `DELETE /orders/0123`, это не означает, что заказ `0123` будет фактически удален из базы данных, содержащей заказы. Такие действия могут просто изменить статус этого заказа на `CANCELED`.

Эти HTTP-методы также должны использоваться для представления *более или менее таковых* действий CRUD. Порой начинающим проектировщикам REST API (а иногда даже и опытным) бывает трудно выбрать, какой HTTP-метод соответствует действию, которое явно не отображается в функцию CRUD. В табл. 5.3 приведены примеры действий, которые могут помочь вам выйти за рамки CRUD.

**Таблица 3.1. HTTP-методы за пределами CRUD**

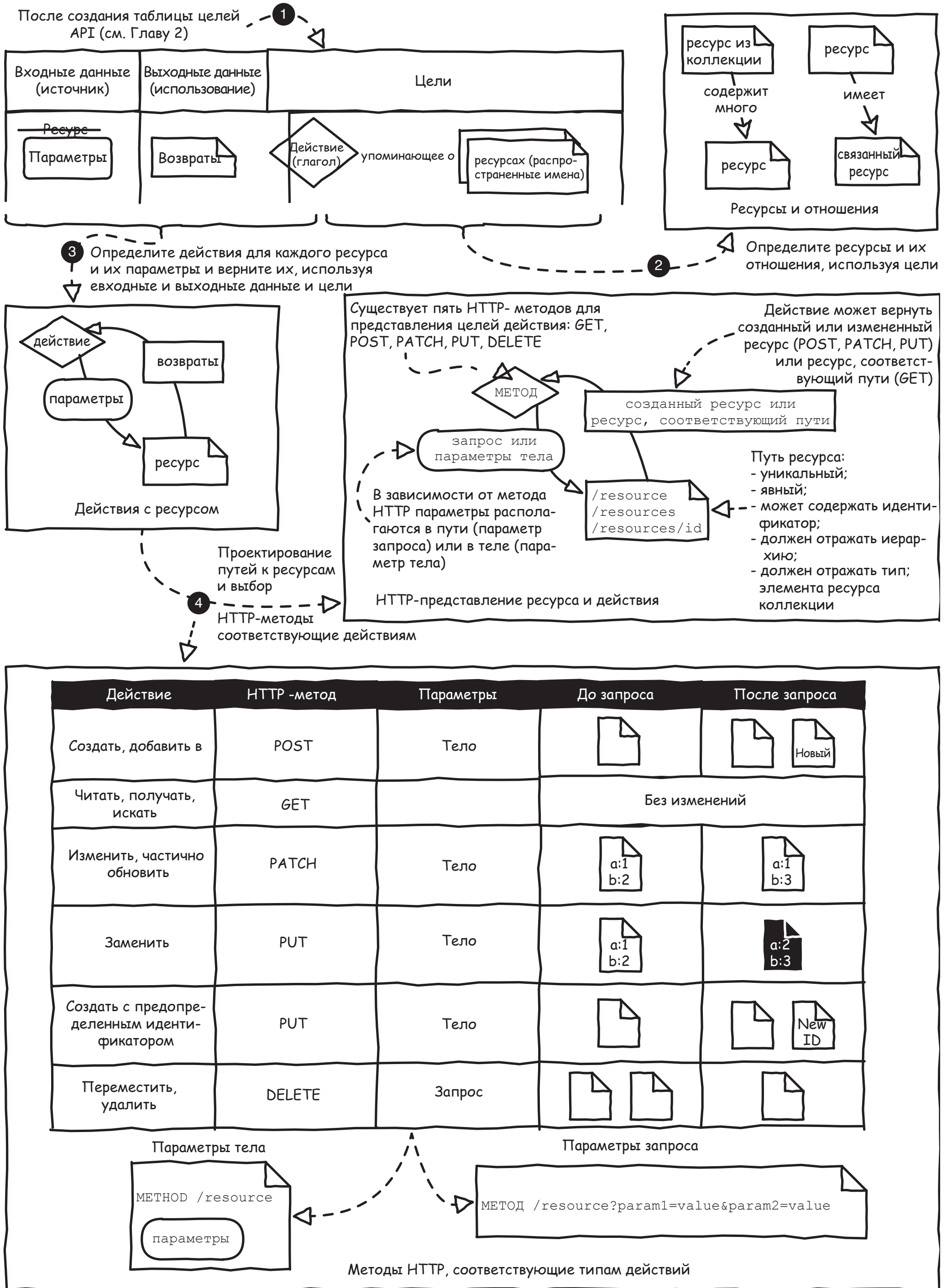
| HTTP-метод | Действие |
|---|---|
| POST (и PUT в создании) | Создать клиента, добавить блюдо в меню, заказать товары, запустить таймер, сохранить запись в блоге, отправить сообщение в службу поддержки, подписаться на услугу, подписать договор, открыть счет в банке, загрузить фотографию, поделиться статусом в социальной сети и т. д. |
| GET | Прочитать клиента, найти французский ресторан, найти новых друзей, извлечь открытые аккаунты за последние три месяца, загрузить подписанный контракт, отфильтровать самые продаваемые книги, выбрать черно-белые фотографии, перечислить друзей и т. д. |
| PATCH/PUT | Обновить клиента, заменить товар в заказе, поменять места в самолете, изменить способ доставки заказа, изменить валюту заказа, изменить лимит дебетовой карты, временно заблокировать кредитную карту и т. д. |
| DELETE | Удалить клиента, отменить заказ, закрыть дело, завершить процесс, остановить таймер и т. д. |

Если вы не можете найти соответствующий HTTP-метод в качестве пары для ресурса и для представления своего действия, по умолчанию можно использовать HTTP-метод `POST` на крайний случай. Мы поговорим об этом подробнее в разделе 3.4.

### *3.2.5 REST API и шпаргалка по HTTP*

Поздравляю! Вы научились преобразовывать цели API в ресурсы и действия REST и представлять их с помощью протокола HTTP. Теперь у вас должен быть общий обзор ресурсов и действий REST API. Давайте подведем итог всему, чему вы уже научились, с помощью шпаргалки, показанной на рис. 3.21. Благодаря ей многое будет проще запомнить!

Помните, в начале этой главы вы видели, что результатом `GET /products/P123` явились некие данные? Теперь мы должны спроектировать их!




| Действие | HTTP -метод | Параметры | До запроса | После запроса |
|---|---|---|---|---|
| Создать, добавить в | POST | Тело | | Новый |
| Читать, получать, искать | GET | | Без изменений | |
| Изменить, частично обновить | PATCH | Тело | a:1 b:2 | a:1 b:3 |
| Заменить | PUT | Тело | a:1 b:2 | a:2 b:3 |
| Создать с предопределенным идентификатором | PUT | Тело | | New ID |
| Переместить, удалить | DELETE | Запрос | | |



**Рис. 3.21. REST API и шпаргалка по HTTP**

## 3.3 Проектирование данных API

Теперь вы знаете, как преобразовать цели API в ресурсы и действия REST и дать им программируемое представление с помощью путей и методов, используя протокол HTTP. Вы также определили параметры действий и результаты. Но идентифицированные вами ресурсы, параметры и возвращаемые результаты описаны нечетко. Как спроектировать эти элементы данных? Процесс проектирования описан на рис. 3.22.

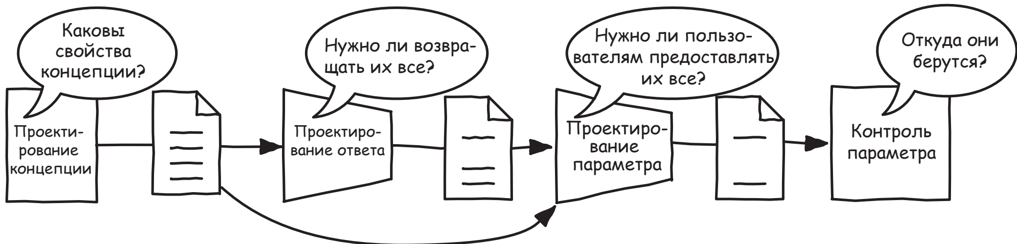



**Рис. 3.22.** Проектирование данных API

Независимо от типа API мы начинаем проектировать данные точно так же, как любое программируемое представление какой-либо концепции – как таблицу базы данных, структуру или объект. Мы просто перечисляем свойства и ориентируемся на потребителя. Потребителям нужно понимать данные, и мы не должны раскрывать внутреннюю работу с помощью дизайна. После того как мы спроектировали основные концепции, можем спроектировать параметры и ответы, адаптируя их. И наконец, мы должны гарантировать, что потребители смогут предоставить все данные, необходимые для использования API.

**ПРИМЕЧАНИЕ.** Для некоторых параметров, таких как бесплатный запрос, который используется при поиске товаров в действии каталога, может потребоваться более простой процесс проектирования.

Как и прежде, самый простой метод, показанный здесь, предназначен для раскрытия основных понятий. Не стесняйтесь адаптировать его или использовать другой метод проектирования программного обеспечения, с которым вы знакомы, пока вы сохраняете настрой и достигаете тех же результатов.

### 3.3.1 Проектирование концепций

Концепции, которые мы определили и превратили в ресурсы REST, будут передаваться через параметры и ответы между потребителем и поставщиком.

Каким бы ни было ее назначение, мы должны позаботиться о разработке такой структуры данных, чтобы предложить ориентированный на потребителя API, как мы это делали при проектировании целей API. На рис. 3.23 показано, как спроектировать такую концепцию в качестве товара.

Мы начнем с перечисления свойств структуры данных и дадим каждому свойству имя. Например, у товара могут быть свойства `reference`,

name и price. Также может быть полезно сообщить покупателю, когда товар был добавлен в каталог (dateAdded) и есть ли он в наличии или нет (unavailable). А как насчет перечисления складов, где можно найти этот товар и его поставщиков? Наконец, у нас может возникнуть желание вернуть более полное описание товара.

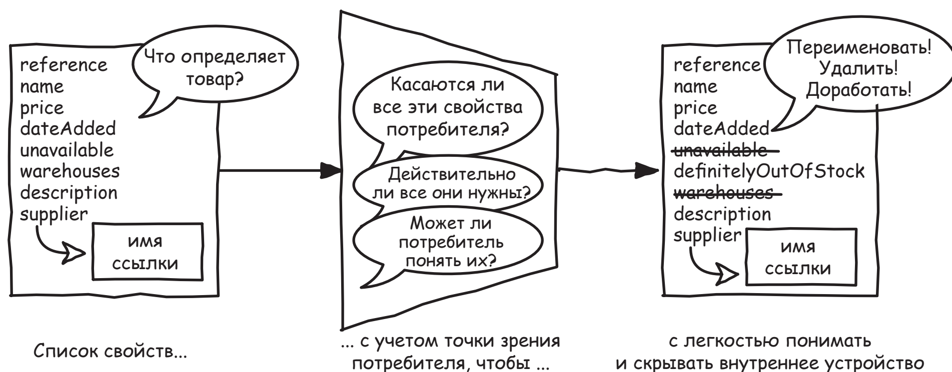



**Рис. 3.23.** Проектирование концепции, ориентированной на потребителя

При перечислении этих свойств следует помнить о том, что вы узнали в главе 2 о точках зрения потребителя и поставщика. Мы должны проанализировать каждую из них, чтобы убедиться, что наш дизайн ориентирован на точку зрения потребителя и не отражает точку зрения поставщика. Это можно сделать, спросив себя, можно ли понять каждое свойство, действительно ли это касается потребителя и действительно ли оно полезно, как показано на рис. 3.23. В нашем примере имена свойств кажутся понятными; мы не использовали непонятных имен, например r и p для обозначения справки и цены. Но если подумать, список warehouses не очень актуален для пользователей, поэтому мы удалим это, а также переименуем свойство unavailable в definitelyOutOfStock, чтобы оно было более явным.

Наиболее важной информацией о свойстве является его название. Чем оно понятнее, тем лучше. Но определения свойства только по имени недостаточно для описания интерфейса программирования, как показано на рис. 3.24.

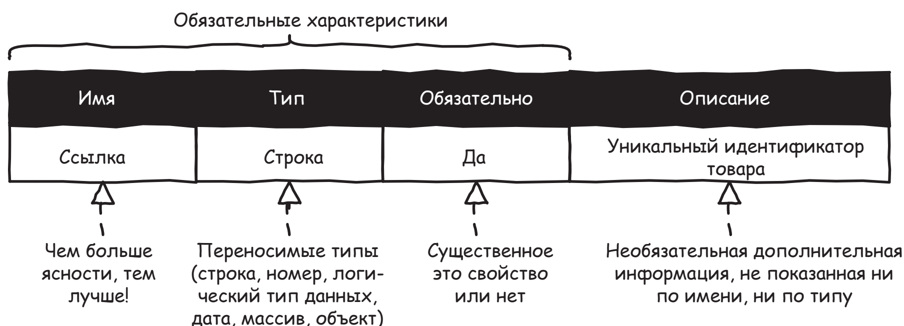



**Рис. 3.24.** Характеристики свойства

Нам также необходимо четко определить тип каждого объекта. Например, каков тип свойства `reference`? Это число или строка? В данном случае это строка. Это существенное свойство, которое всегда должно запрашиваться или возвращаться? В данном случае – да. И последний вопрос: что именно является ссылкой? Ее описание указывает на то, что это уникальный идентификатор, распознающий товар.

Как показано на рис. 3.24, для каждого свойства необходимо собрать следующие характеристики:

- его имя;
- его тип;
- является ли оно обязательным;
- необязательное описание при необходимости.

На рис. 3.25 показан подробный список возможных свойств ресурса товара. В правой части списка также приводится пример документа товара в формате JSON.

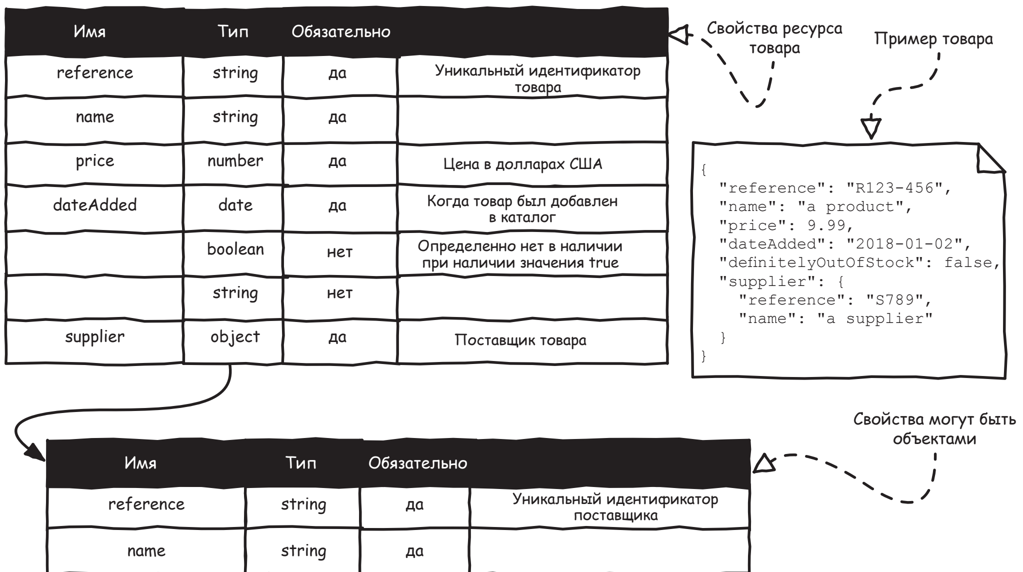



**Рис. 3.25. Свойства ресурса товара**

Итак, товар состоит из обязательных и необязательных свойств. Некоторые из них (например, `reference`, `price`, `definitelyOutOfStock` и `dateAdded`) относятся к базовым типам (например, строка, число, логическое значение или дата). Также могут быть и сложные свойства, например `supplier`, которое является объектом. (Свойство также может быть объектом, содержащим свойства или массив.)

Имя и тип являются наиболее очевидной информацией о свойстве, собираемой при проектировании программного интерфейса. API может использоваться программным обеспечением, написанным на разных языках.

**СОВЕТ.** При выборе типа свойства используйте только переносимые базовые типы данных, общие для языков программирования, такие как строка, число, дата или логическое значение.

Помимо имени и типа свойства, мы также должны знать, должно ли это свойство присутствовать всегда. Указание того, является ли свойство обязательным или необязательным, – аспект проектирования API, о котором часто забывают, но эта информация имеет решающее значение для контекста параметров и ответов для проектировщиков, потребителей и разработчиков, отвечающих за реализацию API. Обратите внимание, что обязательный или необязательный статус свойства может варьироваться в зависимости от контекста. На данный момент мы установим для этого статуса значение «обязательный», только если он является неотъемлемым свойством концепции.

Иногда имени и типа недостаточно для точного описания свойства. Чтобы предоставить дополнительную информацию, которая не может быть явно отражена в имени и типе свойства, может быть полезным добавить описание. Как только мы узнаем, из чего состоят наши концепции, то сможем использовать их в качестве ответов или параметров для наших целей.

### 3.3.2 Проектирование ответов от концепций

Одна и та же концепция может появляться в разных контекстах, как показано на рис. 3.26.

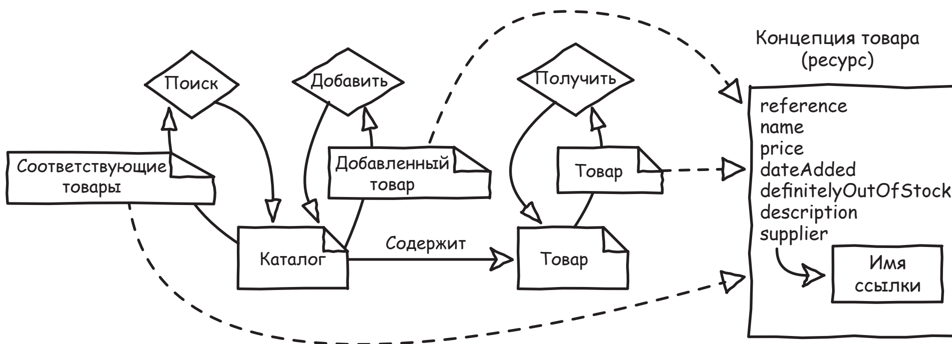



**Рис. 3.26.** Различные представления одной и той же концепции в разных контекстах ответа

Например, действия ресурса каталога «Добавить товар» и «Поиск товаров» возвращают товар (или, в последнем случае, потенциально несколько товаров). Действие «Получить товар» также возвращает товар. Эти разные представления могут не совпадать с теми, что показаны на рис. 3.27.

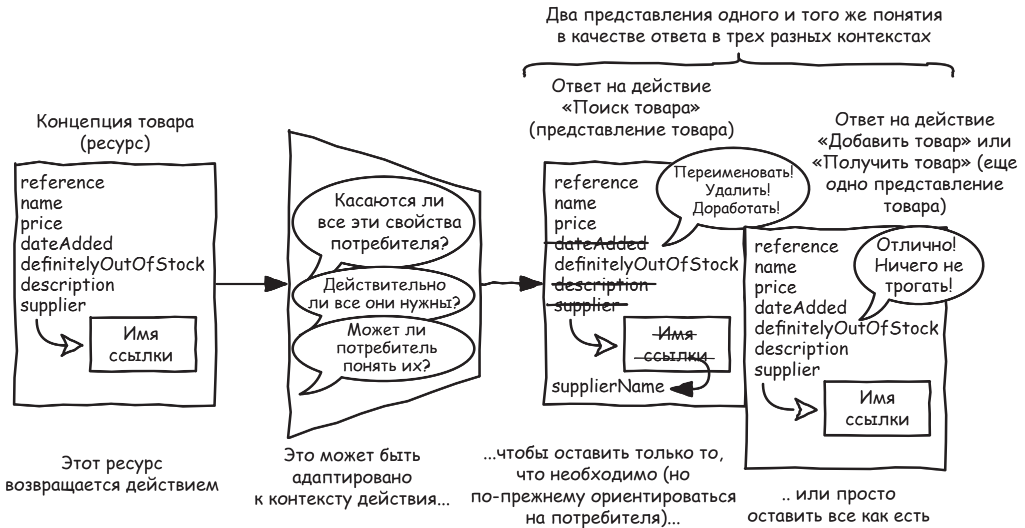



**Рис. 3.27.** Проектирование разных ответов от одной концепции

Если действия «Добавить товар» и «Получить товар» должны вернуть товар целиком, действие «Поиск товаров» может вернуть только ссылку, имя, цену и название поставщика в виде `supplierName`.

При проектировании ответов мы не должны слепо сопоставлять манипулируемый ресурс. Мы должны адаптировать их к контексту, удаляя или переименовывая свойства, а также корректируя структуру данных. А параметры мы проектируем так же?

### 3.3.3 Проектирование параметров из концепций или ответов

Когда мы добавляем, обновляем или заменяем товар, мы передаем некую информацию о товаре в качестве параметра (рис. 3.28).

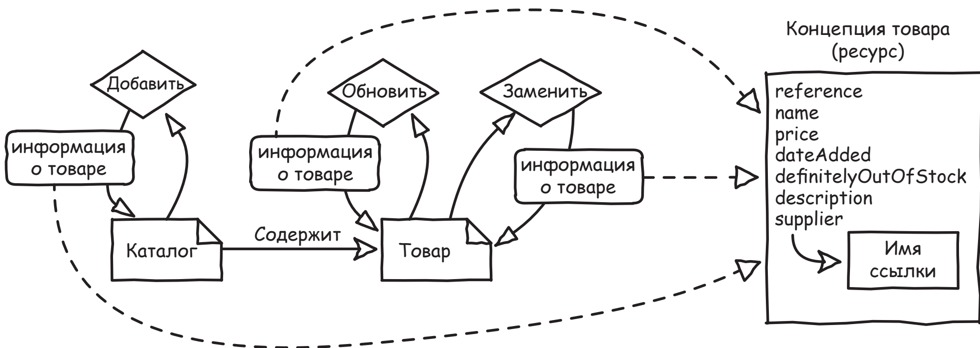



**Рис. 3.28.** Различные представления одной и той же концепции в разных контекстах параметров

Но из чего состоит этот параметр? Или, точнее, *эти* параметры, как показано на рис. 3.29. Параметр информации о товаре, передаваемый в этих трех случаях, может быть разным: они могут не выглядеть как от-

веты, которые мы только что спроектировали, и могут не совсем отражать нашу концепцию товара.

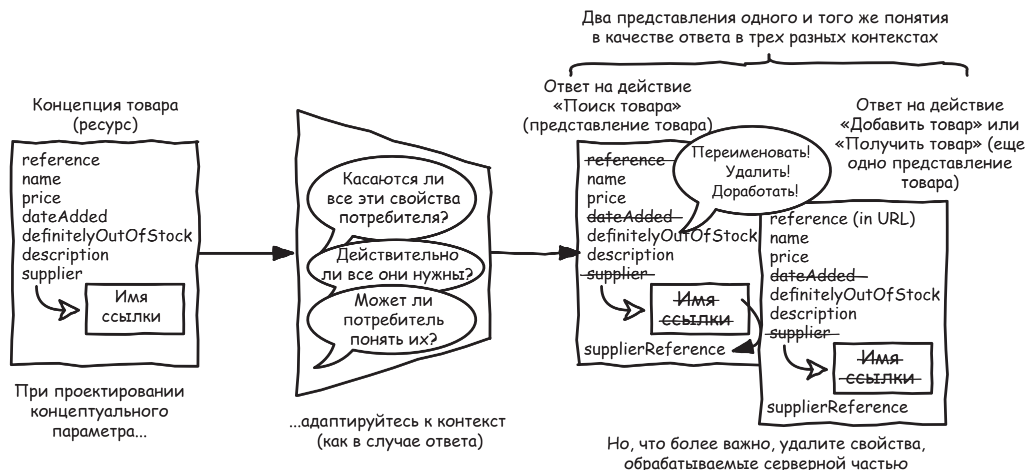



**Рис. 3.29.** **Проектирование различных параметров из единой концепции**

При создании товара ссылка на него генерируется серверной частью, поэтому потребителю не нужно предоставлять ее при добавлении товара. Но нам эта ссылка нужна для обновления или замены товара (обратите внимание, что ссылка будет передана в путь в качестве его параметра: `/products/{reference}`). Во всех этих случаях потребителю не нужно предоставлять свойство `supplier.name` – нужна только ссылка; у серверной части есть способ найти имя поставщика на основе ссылки. Поэтому, чтобы упростить организацию данных, можно удалить объект `supplier` и заменить его на `supplierReference.dateAdded` также генерируется серверной частью при добавлении товара в каталог, поэтому нам это тоже не нужно. Как и в случае с ответами, одна и та же концепция может иметь разные представления в параметрах API в зависимости от контекста (например, создание в сравнении с обновлением).

**ПРИМЕЧАНИЕ.** Параметр должен предоставлять необходимые данные, но не более того. Он не должен включать в себя данные, которые обрабатываются исключительно серверной частью.

Действуя таким образом, мы гарантируем, что параметр информации о товаре содержит только те данные, которые абсолютно необходимы в каждом контексте. А мы уверены в том, что потребитель может предоставить все эти данные?

### *3.3.4 Проверка параметров источника данных*

При добавлении товара в каталог потребители должны иметь возможность легко предоставлять такие данные, как название, цена и описание. А что насчет `supplierReference`? Откуда потребители могут знать такую ссылку? Такие вопросы, вероятно, вам знакомы, потому что при

идентификации целей API мы убедились, что потребители могут предоставить все необходимые входные данные, либо потому что они уже располагают сведениями, либо из-за того, что они могут извлечь их из другой цели API. Но сейчас мы имеем дело с более подробным представлением этих входных параметров.

Потребители должны иметь возможность предоставлять все данные параметра либо потому, что они сами располагают сведениями, либо по той причине, что они могут извлечь их из API. Если данные не могут быть предоставлены, это может быть признаком отсутствующей цели или точки зрения поставщика. Поэтому мы должны еще раз проверить, что все необходимые данные могут быть предоставлены потребителем. Этот процесс проверки, показанный на рис. 3.30, гарантирует, что потребители всегда смогут предоставить данные параметров и что в API нет никаких пробелов.

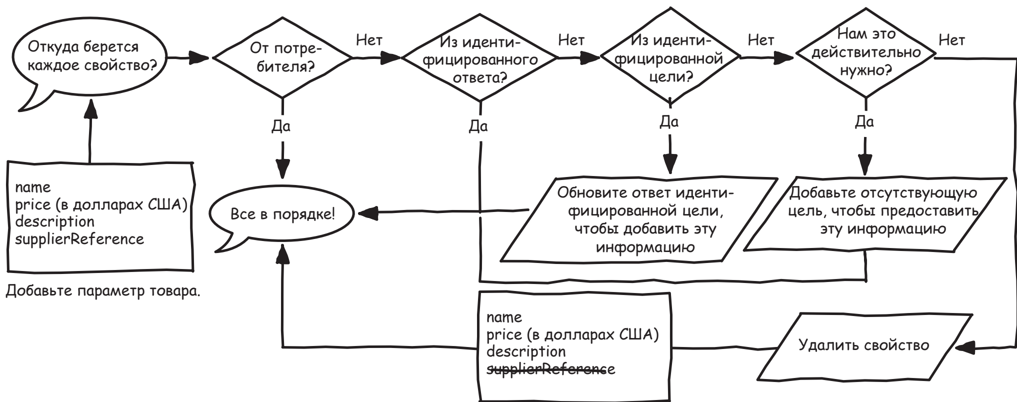



**Рис. 3.30.** Проверяем, что потребители могут предоставить все данные параметра

В этом случае потребители уже могут знать о `supplierReference`, поскольку эта информация указана на этикетке товара или ее можно получить из уже спроектированного нами ответа. Она также может исходить из другой цели; в этом случае нам просто нужно обновить ответ, чтобы добавить эту информацию и предоставить ее потребителю. Возможно, мы просто пропустили цель. В этом случае нам нужно будет добавить ее в таблицу целей API и обработать, как любую другую цель, определив, кто может ее использовать, ее входные и выходные данные и спроектировав ее программируемое представление. Или мы могли бы просто понять, что нам это не нужно.

Мы не будем здесь разгадывать эту загадку. Ее цель – показать, что параметры нужно тщательно проверять и что вы можете обнаружить какие-то недостающие функции, когда познакомитесь с деталями разработки API. Это совершенно нормально; проектирование API – процесс непрерывного улучшения. Шаг за шагом дизайн будет шлифоваться, чтобы стать более точным, полным и эффективным.

### 3.3.5 Проектирование других параметров

А что насчет параметра `free-query` цели «Поиск товаров»? Это строка, в которой может содержаться частичное описание, имя или ссылка. Это необязательный параметр запроса; если не указано иное, возвращаются все доступные товары.

Каким бы ни был параметр – параметр запроса, такой как `free-query`, или даже параметр `body`, не основанный на определенной концепции, – мы делаем то же самое. Мы выбираем ориентированное на потребителя представление и проверяем, может ли тот предоставить все запрошенные данные.

На основании того, что вы уже узнали, можете спроектировать любой базовый REST API. Но иногда есть шанс столкнуться со сложными проблемами проектирования, и придерживаться выбранного представления типа API может быть сложно. В мире проектирования API совершенство не всегда возможно.

## 3.4 Достижение баланса при решении проблем проектирования

Когда вы решите использовать определенный тип API, важно знать, что иногда можно столкнуться с ограничениями. Вы можете изо всех сил пытаться найти представление цели, которая соответствует выбранной модели API, или получить представление, соответствующее модели, но оно будет не таким удобным для пользователя, как вы ожидали. Иногда идеального представления не существует, поэтому вам, как проектировщику API, придется искать компромисс.

### 3.4.1 Примеры компромисса

Сопоставление действий над ресурсами с HTTP-методами и путями – не всегда простая задача. Существуют общепринятые методики, позволяющие обойти такие проблемы, и часто, исследуя различные решения, вы, наконец, можете найти то, которое подходит для выбранной вами модели API. Но иногда это решение может быть неудобным для пользователя.

Возможно, вы заметили, что я тщательно избегал переноса целей, которые были связаны с покупкой товаров пользователем, и сосредоточился на целях, связанных с каталогом. Цели, связанные с управлением каталогами, идеально подходят для демонстрации основ HTTP и REST API. Ресурсы относительно просты для идентификации, и цели с такими действиями, как *add*, *get* или *search*, *update* и *delete* (кто сказал CRUD?) легко сопоставимы с HTTP-методами. Но не всегда все так просто.

Когда пользователи покупают товары, в конце им нужно оформить заказ. Как представить такую цель, когда неочевидно, как перенести ее в пару «путь–HTTP-метод»? Когда проектировщику не удается сопоставить действие над ресурсом REST с каким-либо HTTP-методом, первым вариантом часто является создание ресурса действия (рис. 3.31).

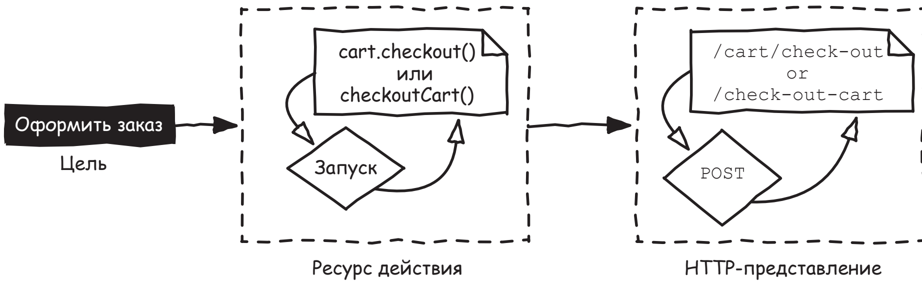



**Рис. 3.31.** Использование ресурса действия

*Ресурс действия* – это не вещь, обозначаемая существительным, а действие, обозначаемое глаголом. Это просто функция. Ресурс действия можно рассматривать как метод класса и, следовательно, он может быть представлен как подресурс ресурса. Если мы решим представить ресурс корзины с помощью пути `/cart`, метод `cart.checkout()` может быть представлен путем `/cart/check-out`. Но мы также могли бы рассматривать его как отдельную функцию `checkoutCart()` и, соответственно, создать ресурс действия `/check-out-cart`. В обоих случаях мы используем HTTP-метод `POST` для запуска ресурса действия.

**ПРИМЕЧАНИЕ.** HTTP-метод `POST` – метод по умолчанию, который используется, когда никакой другой метод не подходит.

Ресурс действия абсолютно не соответствует модели REST, но он работает и полностью понятен потребителям. Давайте посмотрим, сможем ли мы найти решение, которое лучше подходит к модели REST. Например, можно было бы считать, что при оформлении заказа изменяется некий статус (рис. 3.32).

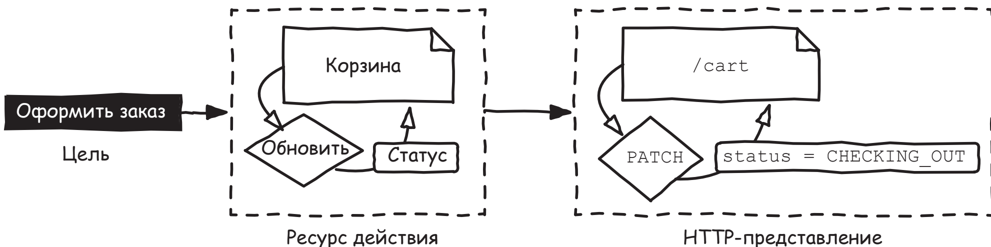



**Рис. 3.32.** Обновление статуса

Ресурс корзины может содержать свойство `status`. Чтобы оформить заказ, мы выполняем обновление с помощью метода `PATCH`, чтобы установить для него значение `CHECKING_OUT`. Такое решение ближе к модели REST, но менее удобно для пользователя по сравнению с ресурсом действия: цель «Оформить заказ» скрыта в обновлении ресурса корзины. Если мы продолжим мозговой штурм, я уверен, что можно будет найти решение, которое полностью соответствует модели REST API.

Давайте вернемся к основам. Нужно задать себе несколько вопросов:

- Что мы пытаемся представить?

- Что происходит в данном случае использования?
- Что происходит, когда мы оформляем заказ?

Итак, создан заказ, содержащий все товары из корзины. И после этого корзина опорожняется. Вот оно! Мы создаем заказ. Следовательно, мы можем использовать запрос `POST /orders`, чтобы создать заказ и оформить его, как показано на рис. 3.33.

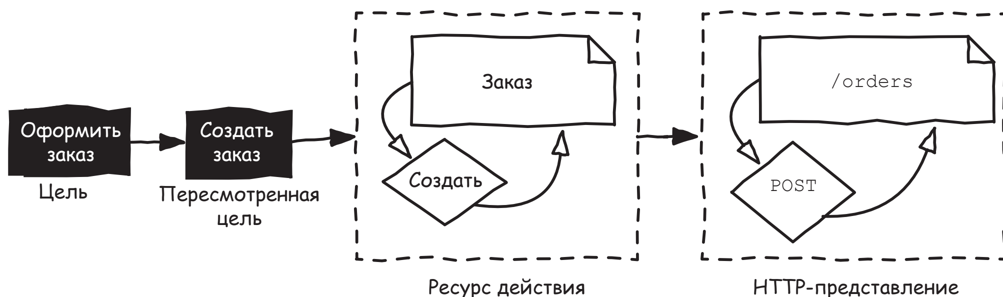



**Рис. 3.33. Соответствие модели REST API**

Это решение полностью соответствует модели REST API, но действительно ли оно удобно для пользователя? Цель этого представления REST может быть неочевидна для всех пользователей.

### *3.4.2 Баланс между удобством для пользователя и соответствием*

Так какой вариант побеждает? Полностью не соответствующий модели REST, но настолько удобный для пользователей `POST /cart/check-out`, или `POST/check-out-cart`? Более соответствующий REST, но немного неуклюжи `PATCH /cart`? Или полностью соответствующий модели REST, но не настолько удобный для пользователя POST /orders? Дело ваше – сделать свой выбор (рис. 3.34) или найти решение получше.

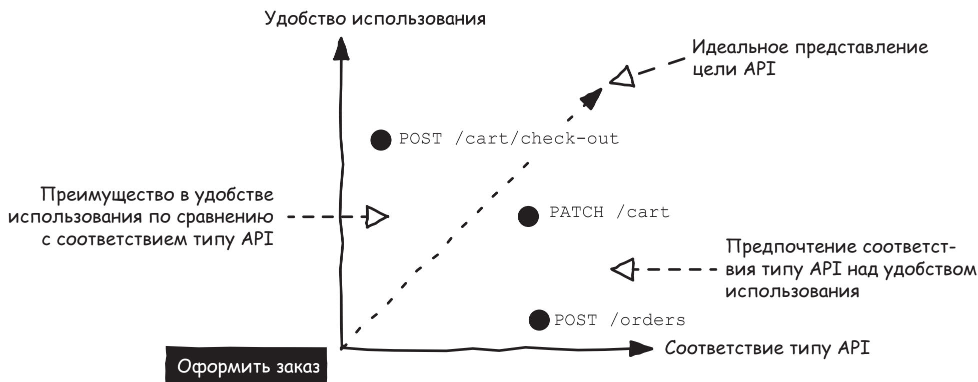



**Рис. 3.34. Баланс между удобством для пользователя и соответствием**

Не всегда можно найти идеальное представление целей API, даже после интенсивного мозгового штурма при помощи всей команды. Иногда вы можете быть не совсем удовлетворены или даже немного разочаро-

ваны дизайном API, над которым работаете. К сожалению, это абсолютно нормально.

Важно освоить азы, лежащие в основе выбранной модели программного интерфейса или стиля API, чтобы иметь возможность находить решения, максимально приближенные к выбранной модели. Но также важно иметь возможность находить разумные компромиссы, чтобы API оставался удобным для потребителя и не слишком отличался от модели API. Эти навыки приходят с практикой в процессе наблюдения за другими API и, самое главное, в ходе общения с вашими потребителями и другими проектировщиками API.

Поздравляю! Теперь у вас должна быть возможность перенести любую цель API в REST API. Но теперь, когда мы рассмотрели, что такое REST API и как создать его на основе таблицы целей API, мы должны исследовать REST помимо сопоставления целей с парами «HTTP-метод–путь». Это важно, потому что REST имеет значение для проектирования любого API.

## 3.5 Почему REST важен при проектировании любого API

Как говорил Т. С. Элиот, «важно путешествие, а не пункт назначения...» Я мог бы объяснить все принципы проектирования API, представленные в этой книге, используя полностью устаревшую модель Xerox Courier RPC, созданную в 1970-х годах, презренную модель SOAP, созданную в конце XX века, теперь уже широко распространенный REST или более поздние gRPC или GraphQL. И это лишь несколько примеров из многих. Как я объяснял в главе 1 (раздел 1.3), были, есть сейчас и всегда будут разные стили программных интерфейсов, позволяющие программному обеспечению обмениваться данными удаленно. У каждого из них есть, были или будут свои особенности, плюсы и минусы, и каждый из них, очевидно, сможет создавать API особого типа. Но независимо от типа проектирования API требует в основном одного и того же мышления.

До этого момента мы рассматривали REST API как API, которые отображают цели в пути и HTTP- методы. Но REST – это намного больше. REST – это широко распространенный стиль API; но, что более важно, он основан на прочной основе *архитектурного стиля REST* – это важно знать при создании любого типа API. Вот почему я выбрал REST в качестве основного примера программного интерфейса для этой книги. Давайте посмотрим, что такое стиль REST и что он означает не только для проектировщиков API, но и для их поставщиков.

### 3.5.1 Знакомство с архитектурным стилем REST

Когда вы набираете URL-адрес, например http://apihandyman.io/about, в адресной строке веб-браузера, он отправляет запрос `GET /about` на веб-сервер `apihandyman.io`. Легко представить, что веб-сервер вернет какой-то статический HTML-документ, хранящийся в его файловой системе, но может быть и по-другому. Содержимое ресурса `/about` может хранится в базе данных. А что происходит, когда веб-сервер социальной сети

получает запрос POST /photos? Сохраняет ли сервер предоставленный файл в качестве документа в папке /photos в файловой системе сервера? Может быть, да. А возможно, и нет. Он также может сохранить это изображение в базе данных.

Браузеры, взаимодействующие с веб-серверами, ничего не знают о таких деталях реализации. Они видят только HTTP-интерфейс, который является лишь абстракцией того, что он может сделать, а не указанием того, как это делается сервером. А как же получается, что веб-браузер может взаимодействовать с любым веб-сервером, реализующим HTTP-интерфейс? Дело в том, что все веб-серверы используют один и тот же интерфейс.

Это часть магии HTTP. Это часть магии REST.

Термин «архитектурный стиль REST» был введен Роем Филдингом в 2000 году в его докторской диссертации «Архитектурные стили и проектирование сетевых программных архитектур». Филдинг разработал этот архитектурный стиль, когда работал над версией протокола HTTP 1.1. В процессе стандартизации HTTP 1.1 он должен был объяснить все – от абстрактных веб-понятий до деталей синтаксиса HTTP – сотням разработчиков. Это и привело к созданию модели REST.

Цель архитектурного стиля REST – облегчить создание эффективных, масштабируемых и надежных распределенных систем. *Распределенная система* состоит из частей программного обеспечения, расположенных на разных компьютерах, которые работают вместе и обмениваются данными по сети, как показано на рис. 3.35.

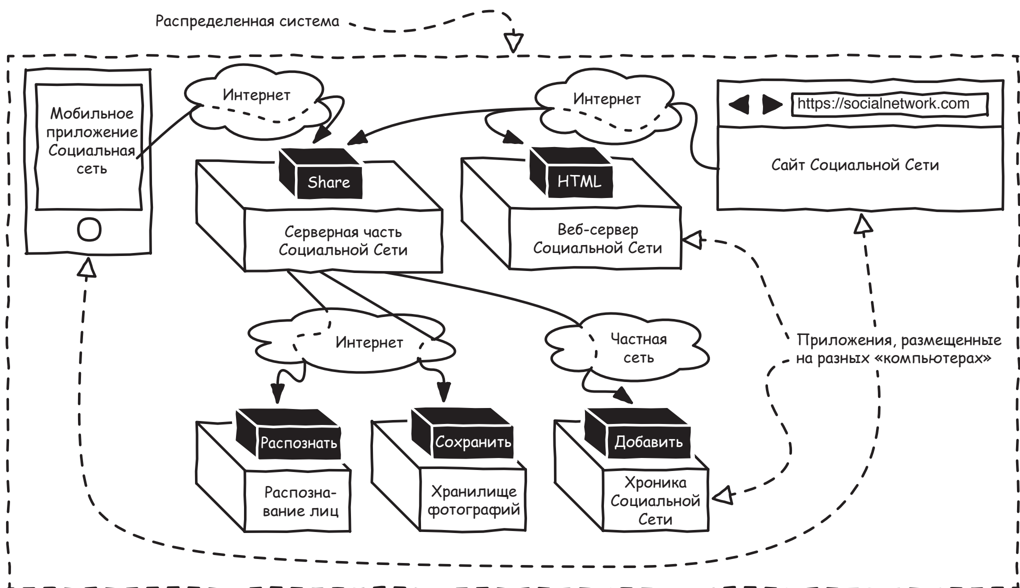



**Рис. 3.35. Распределенная система**

Вам это должно быть знакомым, потому что с самого начала этой книги мы говорили о распределенных системах. Веб-браузер и веб-сервер

образуют такую систему, как и потребитель (например, мобильное приложение), и серверы API. Такие системы должны обеспечивать быстрый обмен данными сети и обработку запросов (*эффективность*), быть способными обрабатывать все больше и больше запросов (*масштабируемость*) и быть устойчивыми к сбоям (*надежность*). Архитектурный стиль REST также направлен на то, что облегчить переносимость компонентов (*повторное использование*), простоту и модифицируемость. Чтобы достичь всего этого – чтобы быть RESTful, – архитектура программного обеспечения должна соответствовать следующим шести ограничениям:

- *модель клиент–сервер* – должно быть четкое разделение проблем, когда такие компоненты, как мобильное приложение и его API-сервер, работают и обмениваются данными друг с другом;
- *отсутствие состояния* – вся информация, необходимая для выполнения запроса, содержится в самом запросе. Клиентский контекст не сохраняется на сервере в сеансе между запросами;
- *кешируемость* – ответ на запрос должен указывать, можно ли его сохранить (чтобы клиент мог повторно использовать его вместо повторного вызова) и как долго;
- *многослойная система* – когда клиент взаимодействует с сервером, он знает только о сервере, а не об инфраструктуре, которая скрывается за ним. Клиент видит только один слой системы;
- *код по запросу* – сервер может передавать исполняемый код клиенту (JavaScript, например). Это ограничение не является обязательным;
- *единообразие интерфейса* – все взаимодействия должны руководствоваться концепцией идентифицированных ресурсов, которыми манипулируют посредством представлений состояний ресурсов и стандартных методов. Взаимодействия также должны предоставлять все метаданные, необходимые для понимания представлений и знания того, что можно сделать с этими ресурсами. Это наиболее фундаментальное ограничение REST. Отсюда же берет свое название этого стиля: Передача состояния представления. В самом деле, использование интерфейса REST состоит из передачи представлений состояний ресурса.

Возможно, это звучит очень пугающе и не относится к проблемам проектирования API, но эти ограничения должны быть понятны любому поставщику API в целом и каждому проектировщику в частности.

### 3.5.2 Влияние ограничений REST на проектирование API

Архитектурный стиль REST изначально создавался как поддержка для описания Всемирной паутины и протокола HTTP, но его можно применять при проектировании любой другой архитектуры программного обеспечения с такими же потребностями. REST API или RESTful API – это API (который в широком смысле включает в себя как интерфейс, так и его реализацию), который соответствует (или, по крайней мере, пытается соответствовать) ограничениям архитектурного стиля REST. Эти ограничения, очевидно, имеют много последствий для REST API и для любого

типа API. Сейчас некоторые из них могут быть несколько трудными для понимания, но мы будем исследовать их на протяжении всей книги, подробно рассматривая различные аспекты проектирования API. Что, если я скажу вам, что мы уже начали исследовать три таких ограничения, возможно, даже не осознавая этого?

Помните точку зрения потребителя, о которой мы говорили в предыдущей главе? Как показано на рис. 3.36, под этим принципом проектирования находятся два ограничения REST.

Исследуя точку зрения потребителя, мы увидели, что поставщик API не должен делегировать свою работу потребителю API – например, включать и выключать магнетрон на кухонном радаре 3000 (см. раздел 2.1). Это пример ограничения *клиент–сервер*.

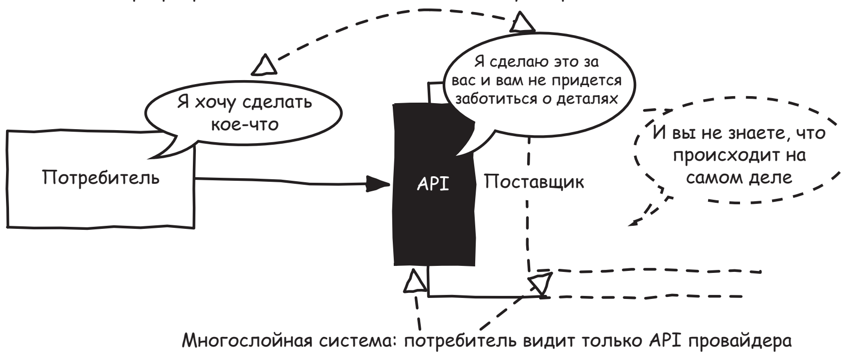



**Рис. 3.36.** Ограничения REST и точка зрения потребителя

Мы также увидели, что потребитель API знает только об API поставщика и не знает, что происходит за пределами этого интерфейса – как в примере с рестораном, где клиенты заказывают еду, не имея понятия о том, что на самом деле происходит на кухне (см. раздел 1.2.2). Вот что означает *многоуровневая система*. Эти два ограничения и ориентация на точку зрения потребителя в целом помогут вам создавать API-интерфейсы, которые легко понять, повторно использовать и развивать.

Мы также затронули тему *однородного интерфейса* в разделах 3.2.3 и 3.2.4, как показано на рис. 3.37.

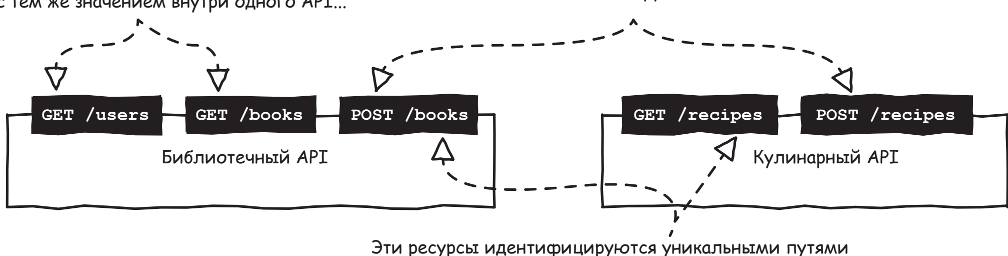



**Рис. 3.37.** Создание API с однородными интерфейсами с использованием протокола HTTP

Каждый ресурс идентифицируется уникальным путем. Внутри одного API и среди нескольких API `POST /resource-A` и `POST /resource-B` имеют одно и то же значение: создавать что-то. Представляя цели с помощью HTTP, используя уникальные пути и, что наиболее важно, стандартизированные HTTP-методы, мы создаем единый интерфейс, который согласуется с самим собой, а также с другими интерфейсами. В главе 6 мы более подробно рассмотрим другие аспекты однородного интерфейса; подробнее поговорим о представлении REST и познакомимся с другими ограничениями (например, отсутствием фиксации состояния, кешируемости и коде по запросу), когда будем учиться проектировать API в следующих главах. В табл. 3.2 приводится сводка всех разделов, описывающих ограничения архитектурного стиля REST.

**Таблица 3.2. Ограничения REST в этой книге**

| Ограничение REST | Упоминание в книге |
|---|---|
| Разделение клиент–сервер | Глава 2, раздел 2.1 |
| Отсутствие фиксации состояния | Глава 5, раздел 5.3.4 |
| Кешируемость | Глава 10, раздел 10.2.2 |
| Многоуровневая система | Глава 1, раздел 1.2.2 |
| Код по требованию | Глава 5, раздел 5.3.2 |
| Однородный интерфейс | Эта глава, разделы 3.2.3 и 3.2.4, и глава 6 |

### *Что почитать о REST, помимо этой книги*

Книга должна предоставить вам достаточно информации обо всех этих концепциях для целей проектирования API, но, если хотите глубже изучить архитектурный стиль REST, есть два документа, которые вы, возможно, захотите прочитать. Первый – это диссертация Филдинга «Архитектурные стили и проектирование сетевых программных архитектур», которую можно найти в свободном доступе на сайте Калифорнийского университета в Ирвайне (UCI), https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. При первом прочтении можете сразу перейти к главе 5, «Передача состояния представления (REST)».

Однако с 2000 года мир изменился, и с тех пор REST использовали, злоупотребляли им или использовали неправильно. Поэтому вам также может быть интересно прочитать публикацию «Размышления об архитектурном стиле REST и «Принципиальное проектирование современной веб-архитектуры», https://research.google.com/pubs/pub46310.html, где описывается история, эволюция и недостатки REST, а также несколько архитектурных стилей, вытекающих из него.

Предупреждаю: эти документы не дают каких-либо конкретных указаний относительно проектирования API.

В следующей главе вы познакомитесь со структурированным способом описания программных интерфейсов, очень похожим на тот, кото-

рый спроектировали мы, узнав, для чего и как описывать API, используя формат описания API.

### Резюме

- REST API представляет свои цели с помощью действий (методы HTTP) над ресурсами (пути).
- При проектировании данных вы должны использовать переносимые данные, такие как объект, массив, строка, число, дата или логический тип данных.
- Одна концепция API может иметь несколько представлений данных в разных контекстах.
- Если параметр содержит данные, которые не могут быть предоставлены потребителями, вы что-то упустили.
- Иногда при проектировании API вы будете разочарованы, и вам нужно будет искать баланс – это совершенно нормально.

# 4 Описание API с помощью формата описания

**В этой главе вы узнаете:**

- что такое формат описания API;
- как описывать REST API с помощью спецификации OpenAPI (OAS).

В предыдущих главах мы изучали, как спроектировать базовый программный интерфейс, используя информацию, собранную в таблице целей API. Мы идентифицировали ресурсы, действия, параметры и ответы для нашего API онлайн-магазина, а также спроектировали данные API. Но все это мы делали, используя диаграммы со стрелками и таблицы.

Такие рисунки и таблицы всегда полезны для мозгового штурма и получения общего представления о том, как можно перенести цели API в программный интерфейс. Но когда речь идет о точном описании программного интерфейса, в особенности его данных, проще и эффективнее использовать структурированный инструмент, такой как формат описания API. Будучи похожим на код и стандартизированным описанием API, он предлагает множество преимуществ:

- значительно облегчает обмен вашим дизайном с любым, кто участвует в вашем проекте;
- его легко могут понять те, кто знает этот формат и инструменты документирования API (среди многих других).

*Спецификация OpenAPI* (OAS) – это популярный формат описания REST API. В этой главе мы рассмотрим его основы, чтобы раскрыть преимущества использования такого формата.

## 4.1 Что такое формат описания API?

*Формат описания API* – это формат данных, целью которого является описание API. На рис. 4.1 показано, что программный интерфейс цели «Добавить товар в каталог» можно описать в простом текстовом файле, используя подобный формат.

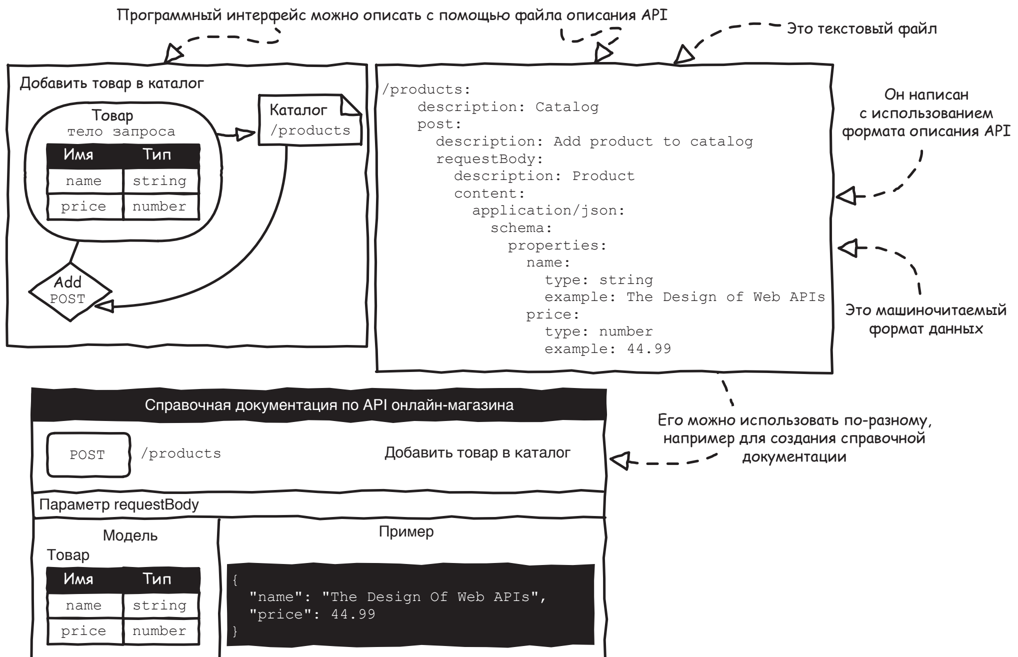



**Рис. 4.1. Описание программного интерфейса с помощью формата описания API**

Этот текстовый файл использует данные, чтобы поведать ту же историю, что и наши таблицы, и нарисованные от руки диаграммы с прямоугольниками и стрелками. Согласно описанию в верхней части документа ресурс `/products` представляет каталог. Он содержит HTTP-метод `POST`, который мы можем использовать для добавления товара в каталог. Этой операции нужно `requestBody`, содержащее название и цену. Описание даже предоставляет дополнительную информацию, такую как пример значений свойств. Такой файл можно быстро написать с помощью простого текстового редактора. Но самое главное – поскольку этот файл содержит структурированные данные, программы могут читать его и с легкостью преобразовывать в нечто другое. Основное использование – создание справочной документации, которая описывает все цели API. Вы можете поделиться им со всеми, кто вовлечен в проект, чтобы они имели представление о создаваемом дизайне.

Звучит интересно, но это только верхушка айсберга. Этот простой текстовый файл использует OAS. Давайте подробнее рассмотрим этот популярный формат описания REST API, прежде чем обсуждать пользу такого формата и то, когда его использовать при проектировании API.

### 4.1.1 Спецификация OpenAPI (OAS)

Спецификация OpenAPI (OAS) – это формат описания REST API, не зависящий от языка программирования. Этот формат продвигается OpenAPI Initiative (OAI), которая «...созданна консорциумом перспективных отраслевых экспертов, осознающих огромную ценность стандартизации описания REST API. Являясь открытой структурой управления в рамках Linux Foundation, OAI нацелена на создание, развитие и продвижение независимого от поставщика формата описания». OAS (https://www.openapis.org) – это формат сообщества; каждый может внести свой вклад через свой репозиторий на сайте GitHub (https://github.com/OAI/OpenAPI-Specification).

#### OAS в сравнении со Swagger

Ранее известный как *спецификация Swagger*, этот формат был передан OAI в ноябре 2015 года и переименован в спецификацию OpenAPI в январе 2016 года. Последней версией (2.0) спецификации Swagger стала OpenAPI 2.0. Она развивалась и на момент написания этой книги ее последняя версия – 3.0.
Изначально спецификация Swagger была создана Тони Тэмом для облегчения автоматизации документирования API и генерации набора средств разработки (Software Development Kit) при работе над продуктами Wordnik1. Эта спецификация была лишь частью фреймворка под названием Swagger API, включавшего в себя такие инструменты, как аннотации кода, генератор кода или пользовательский интерфейс документирования. Все они использовали преимущества спецификации Swagger. Бренд Swagger по-прежнему существует и предоставляет инструменты API с использованием OAS, но имейте в виду, что при поиске информации об этом формате вы можете столкнуться с обоими именами.

[1] Wordnik (https://www.wordnik.com/) – некоммерческая организация, предоставляющая онлайн-словарь английского языка.

На рис. 4.2 показан очень простой документ OAS версии 3.0. Этот документ написан с использованием формата данных YAML.

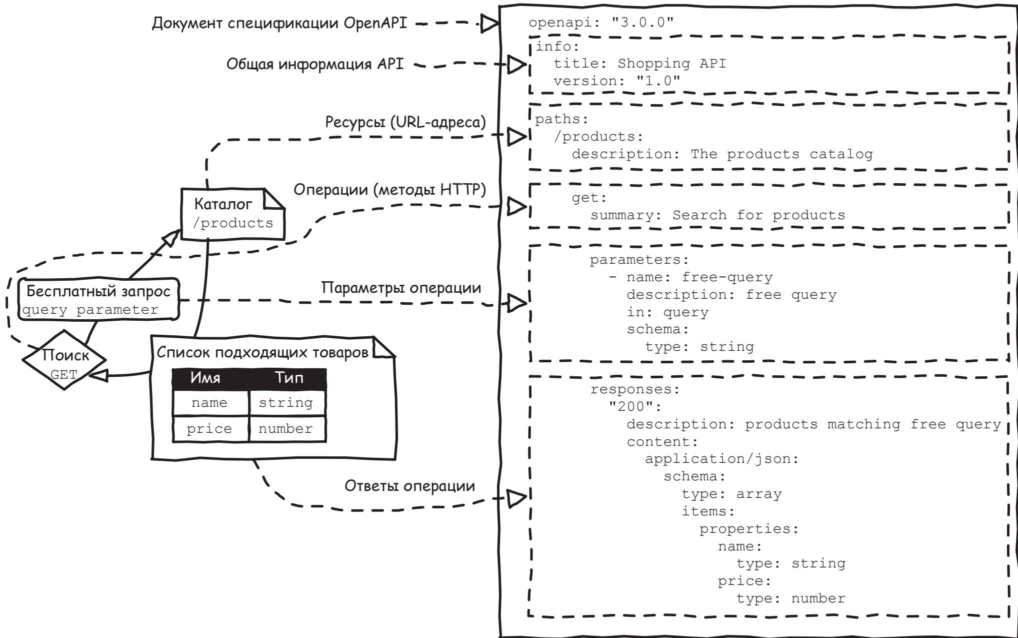
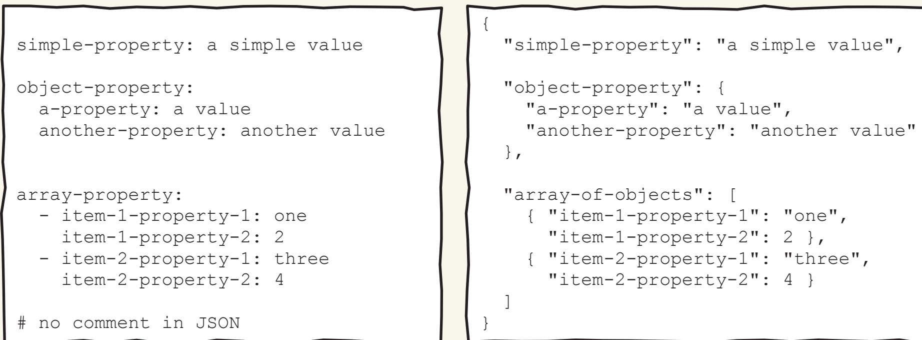



**Рис. 4.2. Документ OAS, описывающий цель «Поиск товаров»**

## YAML

*YAML* (YAML – это не язык разметки) – удобный для восприятия формат сериализации данных. Как и JSON, YAML (http://yaml.org) является форматом данных типа ключ/значение. На рисунке показано сравнение двух этих форматов.



### YAML в сравнении с JSON

Обратите внимание на следующие моменты:

- имена и значения свойств в YAML не заключены в двойные кавычки ("");
- структурные фигурные скобки ({}) и запятые (,) в JSON заменены символами перевода строки и отступами в YAML;
- скобки массива ([]) и запятые (,) заменены черточками (-) и символами перевода строки;
- в отличие от JSON, YAML допускает комментарии, начинающиеся с символа #.

Преобразовать один из этих форматов в другой можно относительно легко. Однако имейте в виду, что при преобразовании документа из формата YAML в JSON комментарии будут утеряны.

Этот базовый документ OAS предоставляет общую информацию об API, такую как его имя и версия. Он описывает доступные ресурсы (идентифицированные по путям) и операции каждого ресурса (или действия, которые вы видели в предыдущей главе), идентифицированные HTTP-методами, включая их параметры и ответы.

Документ OAS может быть написан в формате YAML или JSON, так какой же использовать? Поскольку вы будете писать документы самостоятельно, я рекомендую использовать формат YAML, который, на мой взгляд, легче читать и писать.

Хотя эта книга фокусируется на OAS, существуют и другие форматы описания REST API; наиболее заметными конкурентами OAS являются RAML и Blueprint. Я решил сосредоточиться на OAS не только потому, что ежедневно использую его, но и потому, что он ориентирован на сообщество и очень популярен. Однако обратите внимание, что название этой книги не «Спецификация OpenAPI в действии».

OAS и ее экосистема предлагают множество функций, и, хотя с некоторыми из них мы познакомимся в этой книге, охватить их все будет невозможно. Когда вы ознакомитесь с тем, что представлено в этой книге, я рекомендую вам прочитать документацию по OAS (https://github.com/OAI/OpenAPI-Specification/tree/master/versions) и использовать OpenAPI Map (https://openapi-map.apihandyman.io), инструмент, который я создал, чтобы помочь вам разобраться в этой спецификации.

Вы уже видели, что формат описания API, такой как OAS, позволяет описывать API, используя текстовый файл, содержащий некие структурированные данные. Но вы также можете использовать текстовый процессор или документ электронной таблицы, чтобы сделать то же самое. Я призываю вас не делать этого; посмотрим почему.

### 4.1.2 Зачем использовать формат описания API?

В самом деле, вы могли бы описать программный интерфейс, используя текстовый процессор или электронную таблицу. Вы также можете лег-

ко поделиться таким документом с другими. Но способны ли вы с легкостью управлять его версиями? Можно ли создать из него документацию? Или сгенерировать код? Можете ли вы настроить с его помощью инструменты, связанные с API? Можете ли вы… ? Наверное, я мог бы написать целую страницу подобных вопросов. Использование формата описания API имеет преимущества на протяжении всего жизненного цикла API, особенно на этапе проектирования. Это выгодно не только поставщикам API, но и его потребителям.

### *Эффективное описание API сродни написанию кода*

Документ OAS – это простой текстовый файл, который легко можно сохранить в системе управления версиями, такой как Git, как и код. Таким образом, управлять версиями и отслеживать изменения будет просто.

Документ OAS имеет структуру, которая помогает более эффективно описывать программный интерфейс. Вы должны описать ресурсы, операции, параметры и ответы. Вы можете определить повторно используемые компоненты (например, модель данных), избегая болезненного и рискованного искусства копирования и вставки фрагментов описаний API.

Говоря о написании документа OAS, вы можете использовать свой любимый текстовый редактор, но я рекомендую использовать редактор, который специально предназначен для обработки этого формата. Речь идет об онлайн-редакторе Swagger (http://editor.swagger.io). Он показан на рис. 4.3.

```
1  openapi: "3.0.0"
2  info:
3    title: Shopping API
4    version: "1.0"
5  paths:
6    /products:
7      description: The products catalog
8      get:
9        summary: Search for products
10       parameters:
11         - name: free-query
12           description: free query
13           in: query
14           schema:
15             type: string
16       responses:
17         "200":
18           description: products matching free query
19           content:
20             application/json:
21               schema:
22                 type: array
23                 items:
24                   properties:
25                     name:
26                       type: string
27                     price:
28                       type: number
29
```

GET /products Search for products

Parameters Try it out

Name Description

free-query string (query) free query

free-query - free query

Responses

Code Description Links

200 products matching free query No links

Media type

application/json

Controls Accept header.

Example Value | Schema

name string

price number

Рис. 4.3. Swagger Editor, онлайн-редактор OAS

Поскольку OAS представляет собой машиночитаемый формат, этот редактор предлагает такие функции, как автозаполнение и проверка документов, а правая панель дает полезную визуализацию отредактирован-

ного документа. Чтобы увидеть удобное для восприятия представление структур данных, используемых в качестве параметров или ответов, особенно полезны представления **Model** (Модель) и **Example Value** (Пример значения). Этот редактор является проектом с открытым исходным кодом, который доступен на сайте GitHub (https://github.com/swagger-api/swagger-editor).

Он хорош тем, что для его запуска нужен только браузер, но постоянно скачивать или копировать и вставлять отредактированный файл, чтобы сохранить или открыть его, может быть неудобно. Лично я использую редактор кода Microsoft Visual Studio с расширением Swagger Viewer (https://marketplace.visualstudio.com/items?itemName=Arjun.swagger-viewer), который предоставляет панель предварительного просмотра на базе SwaggerUI и расширение openapi-lint (https://marketplace.visualstudio.com/items?itemName=mermade.openapilint), обеспечивающее автозаполнение и проверку. Эта конфигурация предоставляет те же возможности, что и онлайн-редактор, и вы работаете со своими файлами напрямую.

Обратите внимание, что существуют инструменты проектирования API, которые позволяют описывать программный интерфейс без написания кода. Некоторые из них предлагают интересные функции, такие как совместная работа. Если вы хотите использовать такой инструмент – отлично; просто убедитесь, что вашу работу можно экспортировать в известный и используемый формат описания API. Но даже если такие инструменты и существуют, все же стоит знать, как написать документ OAS. Для использования такого формата практически ничего не нужно, и однажды вы можете захотеть создать собственный инструментарий.

### *Легкий обмен описаниями API и документирование API*

Документом OAS можно легко поделиться с другими даже за пределами вашей команды или компании, чтобы получить отзыв о своем дизайне. В отличие от определенного внутреннего формата, известного лишь немногим, формат OAS очень распространен. Люди могут импортировать документ в онлайн-редактор Swagger Editor или множество других инструментов API. В качестве альтернативы, чтобы не мешать никому самим документом OAS, вы можете предоставить доступ к готовой к использованию и удобной для восприятия визуализации. Документ OAS можно использовать для создания справочной документации по API, в которой показаны все доступные ресурсы и операции. Для этого можно использовать пользовательский интерфейс Swagger (https://github.com/swagger-api/swagger-ui), который показывает документ OAS как в правой панели редактора Swagger Editor.

Есть и другие инструменты. Например, в качестве альтернативы Swagger UI можно использовать такой инструмент, как ReDoc (https://github.com/Rebilly/ReDoc), также с открытым исходным кодом. Он показан на рис. 4.4.

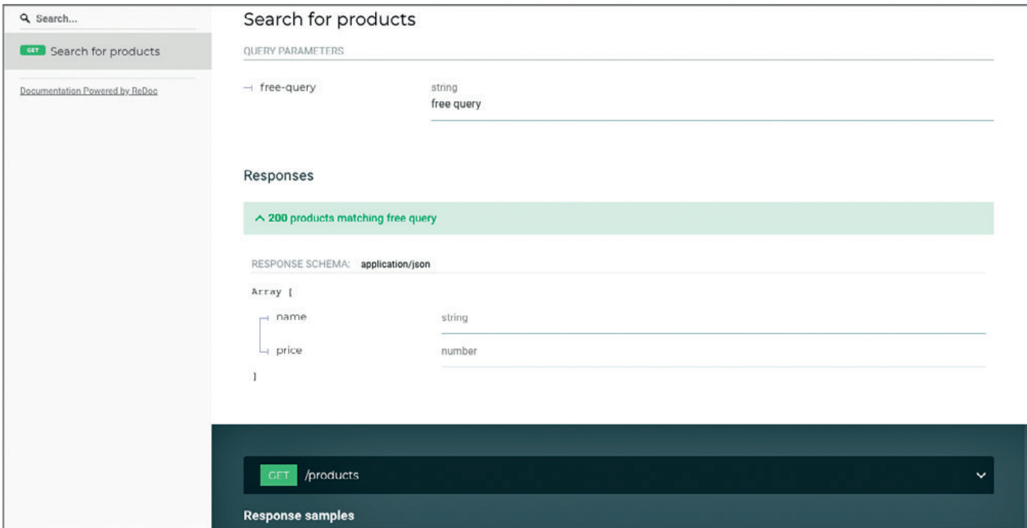



Рис. 4.4. Документ OAS в ReDoc

И последнее замечание: для создания документации по API вы узнаете о расширенном использовании OAS в главе 12.

### *Генерация кода и не только*

Как только API будет описан с помощью формата описания API, из него можно частично сгенерировать код реализации. Вы получите пустой костяк исходного кода, который также можно использовать для создания рабочего макета. Потребители также могут воспользоваться преимуществами таких машиночитаемых описаний API для генерации кода для использования API. И такой формат также может использоваться инструментами тестирования API или безопасности, а также многими другими инструментами, связанными с API. Например, большинство *решений для API-шлюзов* (прокси, созданные для предоставления доступа и защите API) можно настроить с помощью файла описания API, такого как документ OAS.

**СОВЕТ.** Посетите сайт https://openapi.tools/, на котором представлен список инструментов на базе OAS, чтобы получить представление об экосистеме OAS.

Эти примеры показывают, что формат описания API более эффективен, чем текстовый процессор или документ электронной таблицы. И документы OAS можно использовать множеством других способов.

## 4.1.3 Когда использовать формат описания API

Однако, прежде чем приступить к описанию API с использованием формата описания API, необходимо убедиться, что вы делаете это в нужное время, как показано на рис. 4.5.

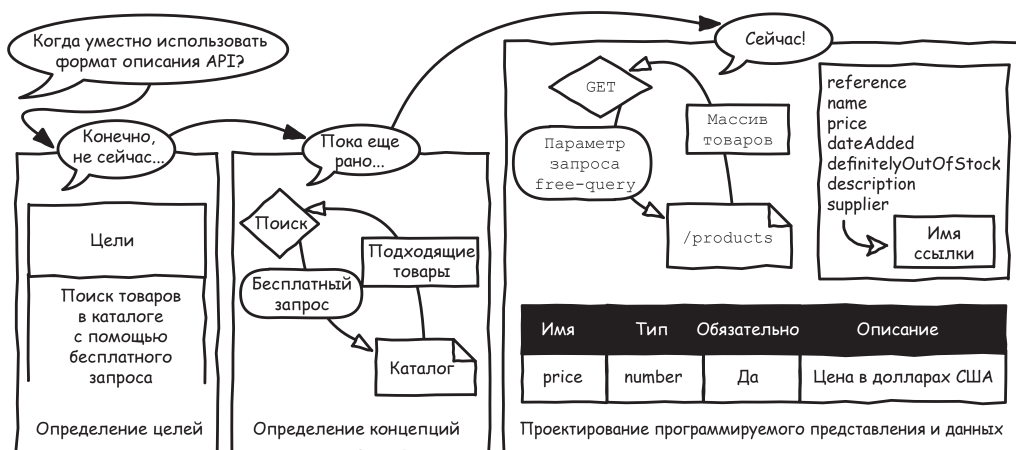



**Рис. 4.5.** Когда использовать формат описания API

Формат описания API создан для описания программного интерфейса. Поэтому его нельзя использовать при идентификации целей API. Как вы узнали из главы 3, проектировать программный интерфейс, не имея четкого представления о том, что должен делать AP, – ужасная идея! И формат описания API не должен использоваться при идентификации концепций, стоящих за целями; для этого еще слишком рано. Конечно, это первый шаг проектирования программного интерфейса, но на этом этапе вы все еще не имеете дело с реальным интерфейсом. Однако формат описания API определенно должен использоваться при проектировании программируемого представления целей и концепций, а также данных.

При проектировании REST API вы можете приступить к использованию OAS, когда проектируете пути к ресурсам и выбираете HTTP-методы, описывающие действия. Можно создать минимальный файл, содержащий только эти элементы. Когда это будет сделано, вы можете заполнить документ, описав данные API. Как вы увидите в следующем разделе, описывать все это будет гораздо проще и эффективнее, используя формат описания API, нежели рисуя таблицы, стрелки и прямоугольники. Но не забывайте, что все те, кто участвует в проекте и кому на самом деле необходимо увидеть дизайн API, могут быть не знакомы с напоминающим код форматом, таким как OAS, поэтому всегда предоставляйте способ получить удобное для восприятия представление файла, над которым вы работаете. Учитывая это, давайте посмотрим, как описать ресурсы и действия REST API с помощью OAS.

## 4.2 Описание ресурсов и действий API с помощью OAS

Как показано на рис. 4.6, когда мы перенесли цели API в программный интерфейс в главе 3, мы идентифицировали ресурсы и действия и представляли их с помощью путей и HTTP-методов.

**Рис. 4.6. От рисунка к документу OAS**

Мы использовали прямоугольники и стрелки для описания этого программного интерфейса, но это также можно сделать и более структурированным способом с помощью OAS. Полученный документ будет содержать точно такую же информацию, что и соответствующий рисунок, но на этот раз информация будет представлена в виде данных в структурированном документе. Давайте начнем с документа, описывающего API онлайн-магазина.

### 4.2.1 Создание документа OAS

В приведенном ниже листинге показан минимальный, но допустимый документ OAS. Этот документ написан с использованием OAS версии 3.0.0. Он описывает API под названием Shopping API в версии 1.0.

**Листинг 4.1. Минимальный, но допустимый документ OAS**

```
openapi: «3.0.0» ①
info: ②
  title: Shopping API
  version: «1.0»
paths: {} ③
```

① **Версия OAS.**
② **Общая информация API.**
③ **Пустые пути**

Структура документов OAS может меняться от одной версии к другой, поэтому парсеры используют версию `openapi` для соответствующей адаптации парсинга. Обратите внимание, что номера версий спецификации (`openapi`) и API (`info.version`) должны быть заключены в кавычки. В противном случае парсеры OAS будут рассматривать их как числа и проверка документа будет неудачной, поскольку эти два свойства должны быть строками.

В листинге свойство `paths` показано только для того, чтобы создать документ. (Если оно будет отсутствовать, парсер сообщит об ошибке.) Свойство `paths` содержит ресурсы, доступные для этого API. Сейчас мы можем установить для него значение `{}` – так описывается пустой объект в YAML. Непустой объект (например, `info`) не нуждается в фигурных скобках. Далее мы начнем заполнять свойство `paths`, добавив ресурс.

### *4.2.2 Описание ресурса*

Как показано на рис. 4.7, работая над целями «Поиск товаров» и «Добавить товар в каталог», мы идентифицировали ресурс каталога. Мы решили представить его с помощью пути `/products`.

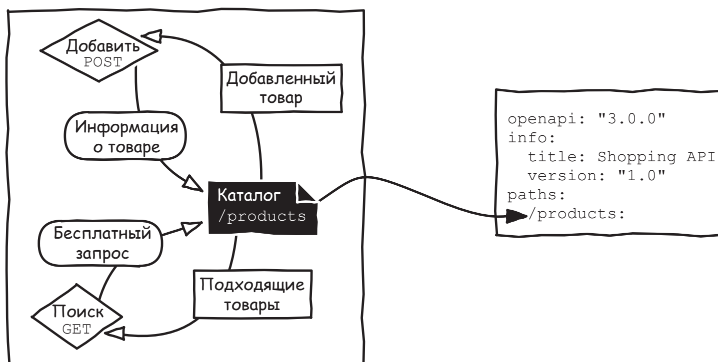



**Рис. 4.7. Добавление ресурса в документ OAS**

Чтобы описать этот ресурс в документе OAS, мы должны добавить путь `/products` в свойство `paths`, как показано на рис. 4.7 (не забудьте удалить пустые фигурные скобки!). Мы также опишем, что это за ресурс (`The products catalog`), используя свойство ресурса `description`, как показано в приведенном ниже листинге.

**Листинг 4.2. Описание ресурса**

```
openapi: «3.0.0»
info:
  title: Shopping API
  version: «1.0»
paths: ①
  /products: ②
    description: The products catalog ③
```

① **Ресурсы API.**

② **Путь к ресурсу.**

③ **Описание ресурса.**

Свойство `description` не является обязательным, но описание ресурсов вашего API будет полезно на протяжении всего жизненного цикла API. Это как когда вы пишете код: код может быть понятен сам по себе, но комментарии или аннотации JavaDoc, PHPDoc, JSDoc или <ваш люби-

мый язык> Doc по поводу его использования всегда будут приветствоваться другими людьми, которые читают ваш код или сгенерированную из него документацию.

Поскольку API по определению будет применяться другими лицами, очень важно использовать возможности документирования форматов описания API. На этом этапе проектирования особенно полезно сохранять связь между своей предыдущей работой, такой как таблица целей API или идентификация концепции, и программируемым представлением. Это также может помочь людям, с которыми вы делитесь этим дизайном, легче понять его (`/products` в качестве каталога может быть очевиден не для всех).

Ресурс, описанный в документе OAS, должен содержать какие-то операции. В противном случае документ будет недействителен. Ресурс каталога используется двумя целями: «Поиск товаров» и «Добавить товар». Давайте посмотрим, как описать их в качестве операций в документе OAS.

### 4.2.3 Описание операций в ресурсе

Мы можем добавить к нашему документу действие, чтобы предоставить всю информацию для каждой цели, которую мы идентифицировали к концу раздела 3.2. В случае с каждой из них мы знаем, какой HTTP-метод она использует, и у нас есть текстовое описание ее входных и выходных данных, как показано в левой части рис. 4.8.

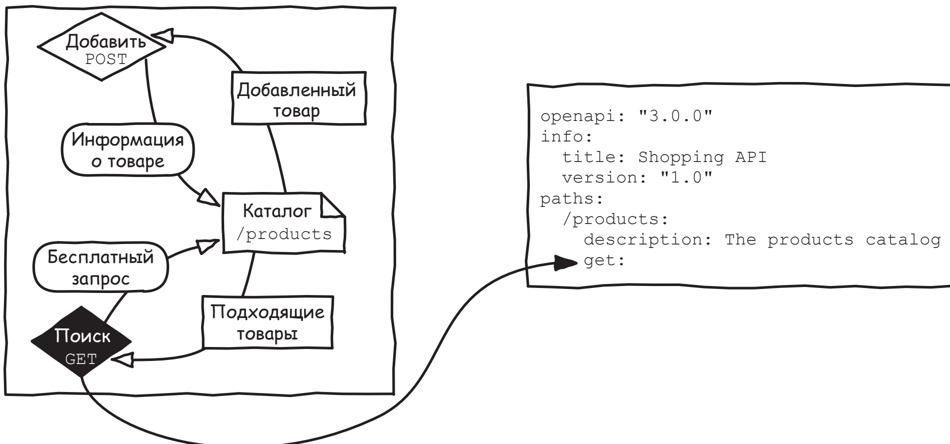



**Рис. 4.8.** Добавление действия к ресурсу

Для представления поиска товаров мы решили использовать HTTP-метод `GET` для ресурса каталога, представленного путем `/products`. Это действие использует параметр *бесплатный запрос* и товары, соответствующие запросу. Чтобы добавить это действие к ресурсу `/products`, мы используем свойство `get`. Мы также можем использовать функции документирования, чтобы предоставить больше информации об операции `GET /products`. Мы установим для свойства `summary` значение `Search for`

products, а для свойства description – Search for products in catalog using a free query parameter, как показано в приведенном ниже листинге.

**Листинг 4.3. Описание действия над ресурсом**

```
openapi: «3.0.0»
info:
  title: Shopping API
  version: «1.0»
paths:
  /products: ①
    description: The products catalog
    get: ②
      summary: Search for products ③
      description: |  ④
      Search for products in catalog
      using a free query parameter
```

① Ресурс.
② HTTP-метод действия.
③ Краткое описание действия.
④ Длинное описание действия.

Свойство summary – это краткое описание действия без подробностей. Цель, определенная на таблице целей API, обычно идеально подходит для использования здесь. Есть также свойство description, которое можно взять для предоставления более подробного описания действия. Здесь мы применяем его, чтобы указать, что это действие использует параметр бесплатного запроса (using a free query parameter). Обратите внимание, свойство description является многострочным. Это особенность YAML: чтобы быть многострочным, свойство string должно начинаться с символа вертикальной черты (|).

В документе OAS операция должна описывать по крайней мере один ответ в свойстве responses, как показано в листинге 4.4. На данный момент мы будем использовать этот обязательный ответ, чтобы предоставить неформальное описание результатов поиска товаров. Мы добавим свойство responses, содержащее ответ «200» (для кода состояния HTTP 200 OK), где значение свойства description – Products matching free query parameter.

**Листинг 4.4. Описание ответов на действия**

```
openapi: «3.0.0»
info:
  title: Shopping API
  version: «1.0»
paths:
  /products:
```

```
    description: The products catalog
    get:
      summary: Search for products
      description: | Search for products in catalog
        using a free query parameter
      responses: ①
        "200": ②
          description: | ③
          Products matching free query parameter
```

① **Список ответов.**
② **Код ответа 200 OK.**
③ **Описание ответ.**

Как уже упоминалось, возможные ответы действия описаны в свойстве responses. Каждый ответ идентифицируется кодом состояния HTTP и *должен* содержать свойство description. Свойство «200» означает HTTP-статус «200 OK», который сообщает потребителю, что все прошло нормально. (Вы заметили кавычки вокруг кода состояния? Они необходимы, потому что имена свойств YAML должны быть строками, а 200 – это число). Свойство ответа description гласит, что, если все прошло нормально, возвращаются Products matching free query parameter. Мы рассмотрим возможные ответы и коды состояния HTTP, возвращаемые действием, более подробно в главе 5.

Теперь, когда мы добавили это действие к ресурсу /products, наш документ OAS является допустимым. Но мы еще не закончили. Для этого ресурса есть второе действие: добавить товар. Мы выбрали HTTP-метод POST для представления действия «Добавить товар» в ресурсе каталога. Он берет какую-то информацию о товаре и возвращает его добавленным в каталог (см. рис. 4.9).

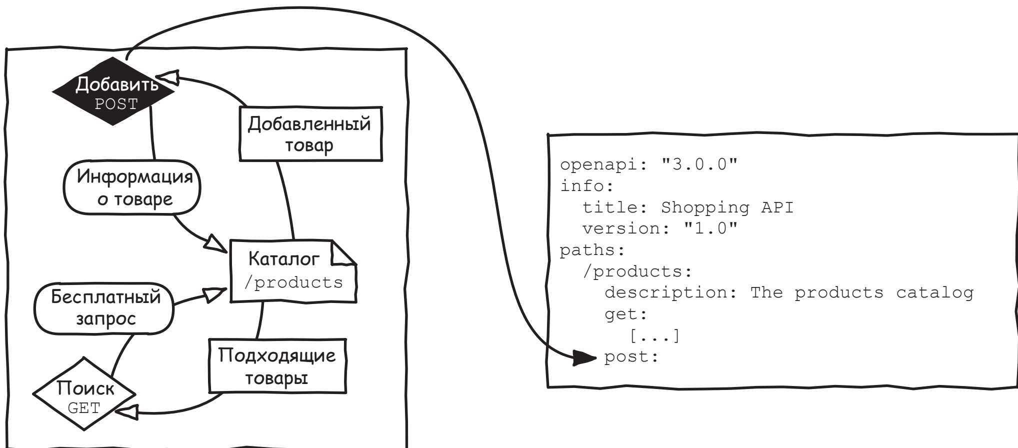



**Рис. 4.9. Добавляем еще одно действие к ресурсу**

Чтобы добавить это действие, мы поступаем точно так же, как и ранее. В приведенном ниже листинге показано, как это сделать.

**Листинг 4.5. Описание еще одного действия**

```
openapi: "3.0.0"
info:
  title: Shopping API
  version: «1.0»
paths:
  /products: ①
    description: The products catalog
    get:
      summary: Search for products
      description: |
        Search for products in catalog
        using a free query parameter
      responses:
        "200":
          description: |
          Products matching free query parameter
post: ②
  summary: Add product ③
  description: | ④
      Add product (described in product info
      ➥ parameter) to catalog
responses: ⑤
  "200": ⑥
    description: | ⑦
      Product added to catalog
```

① **Ресурс.**
② **HTTP-метод действия.**
③ **Краткое описание действия.**
④ **Длинное описание действия.**
⑤ **Список ответов.**
⑥ **Ответ 200 OK.**
⑦ **Описание ответа 200 OK.**

Мы добавляем свойство `post` внутри объекта, описывая ресурс каталога, указанный с помощью пути `/products`. Мы устанавливаем для его свойства `summary` значение `Add product`, а для свойства `description` – значение `Add product (described in product info parameter) to catalog`. Мы добавляем свойство «`200`» в `responses` и устанавливаем для его свойства `description` значение `Product added to catalog`.

Операция `post` ресурса `/products` теперь описана в документе OAS. Любой, кто просматривает этот документ, может сказать, что делает эта операция, прочитав сводку, описание и описание ответа. Как показано на рис. 4.10, этот документ содержит ту же информацию, которую мы идентифицировали в разделе 3.2.

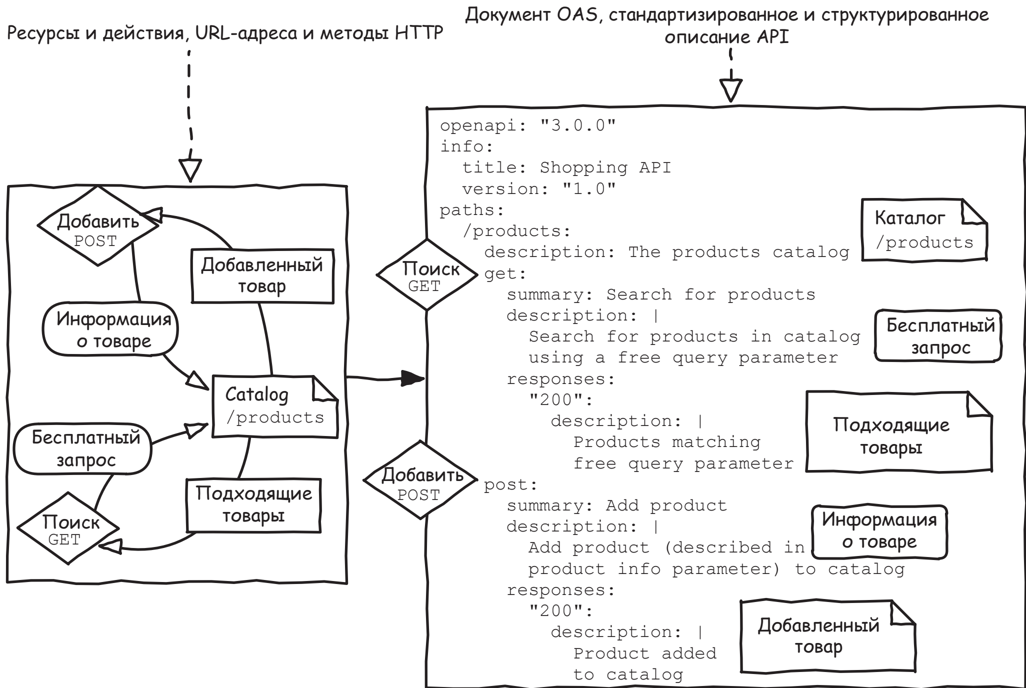



**Рис. 4.10.** Документ OAS, соответствующий исходному рисунку

Поздравляю! Теперь вы знаете основы, для того чтобы описать ресурс и его действия с помощью OAS. Даже если оно еще не полное, такое описание уже предоставляет интересную информацию об API. У нас есть формальное и структурированное описание пути к ресурсу и методов HTTP, и мы можем определить, какой цели какое действие соответствует и как они работают, благодаря свойствам `description`.

Но этот документ содержит только смутное описание входных и выходных данных каждой операции. В предыдущей главе мы подробно спроектировали их. Давайте теперь посмотрим, как завершить этот документ, описав эти данные.

## 4.3 Описание данных API с помощью OpenAPI и JSON Schema

В предыдущей главе, когда мы проектировали программный интерфейс, соответствующий идентифицированным целям, мы не останавливались после проектирования путей к ресурсам и выбора HTTP-методов. В разделе 3.3 мы полностью описали параметры действий и ответы, включая описания организации и свойств данных (рис. 4.11).

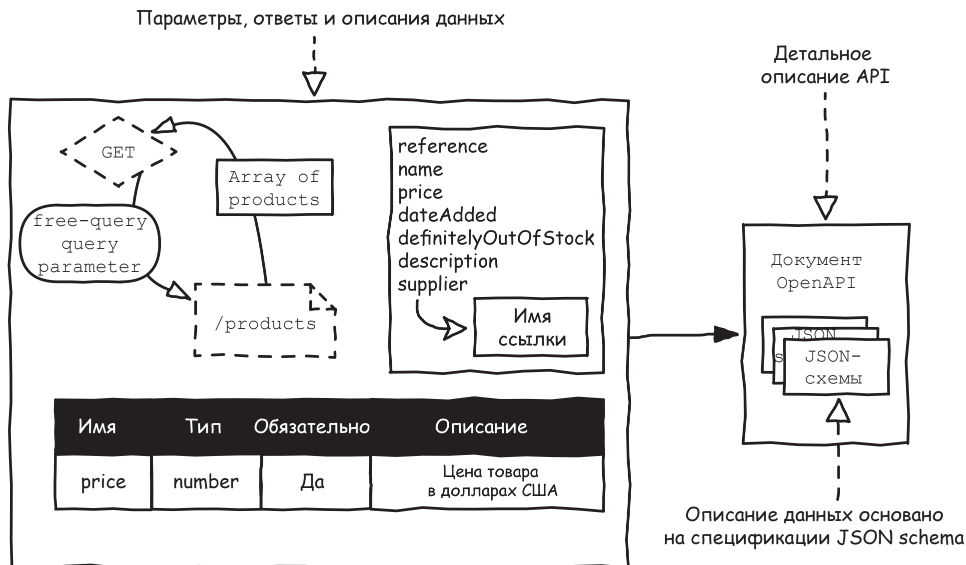



**Рис. 4.11.** От рисунков и таблиц к подробному документу OAS

OAS использует спецификацию JSON Schema (http://json-schema.org) для описания всех данных – например, параметров запроса, параметров тела или тела ответа. JSON Schema стремится описывать форматы данных понятным для человека и машин способом. Его также можно использовать для проверки документов в формате JSON по схеме JSON, описанию данных, созданному с помощью JSON Schema. Этот формат можно использовать независимо от OAS для описания и проверки любого типа данных JSON.

**ПРИМЕЧАНИЕ.** OAS использует адаптированное подмножество JSON Schema. Она использует не все функции JSON Schema, и некоторые определенные функции OAS были добавлены в это подмножество.

В этой главе *JSON Schema* ссылается на спецификацию JSON Schema, тогда как схема JSON представляет собой фактическую схему, описание данных. Обратите внимание на разницу в написании. Давайте посмотрим, как описать данные API, используя OAS и JSON Schema. Начнем с параметра запроса «Поиск товаров».

## 4.3.1 Описание параметров запроса

Для поиска товаров пользователи API должны предоставить параметр *бесплатного запроса*, чтобы указать, что они ищут (рис. 4.12). В предыдущей главе мы решили, что это будет параметр запроса с именем `free-query`. Для поиска товаров с использованием API потребитель должен выполнить запрос `GET /products?Free-query={free query}` (например, `GET /products?free-query=book`).

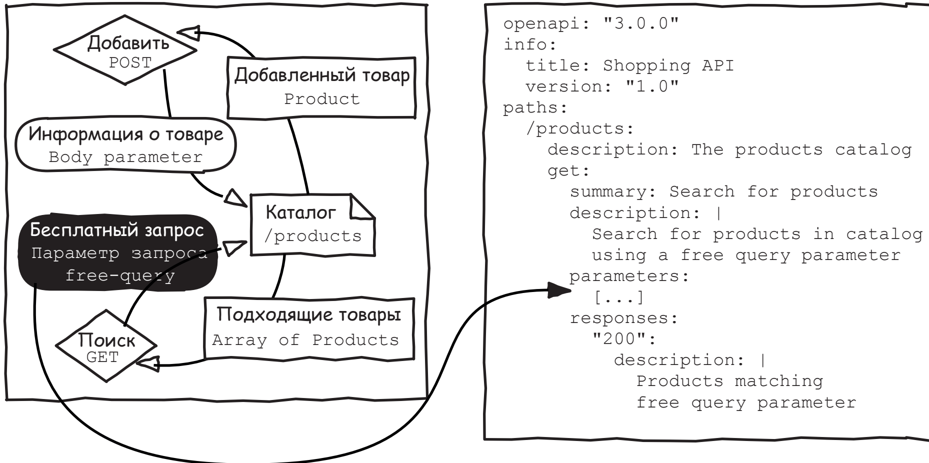



**Рис. 4.12. Параметр запроса для поиска товаров**

Чтобы описать этот параметр, мы добавляем свойство parameters внутри операции get ресурса /products, как показано в приведенном ниже листинге.

**Листинг 4.6. Описание параметров**

```
openapi: "3.0.0"
info:
  title: Shopping API
  version: "1.0"
paths:
  /products: ①
    get: ②
      summary: Search for products
        description: | Search for products in catalog
          ➥ using a free query parameter
          parameters: ③
          [...]
      responses:
        "200":
          description: | Products matching free query
          ➥ parameter
```

① **Ресурс.**
② **Действие.**
③ **Список параметров действия (за исключением тела).**

Когда действию над ресурсом нужны параметры, отличные от параметров тела, они описываются в свойстве действия parameters. В этом случае, чтобы описать параметр, мы устанавливаем для его имени значение free-query, как показано в приведенном ниже листинге.

**Листинг 4.7. Описание параметра запроса**

```
parameters:
  - name: free-query ①
    description: |  ②
      A product's name, reference, or partial
      ➥ description
      in: query ③
      required: false ④
      schema: ⑤
        type: string ⑥
```

① **Имя параметра.**
② **Описание параметра.**
③ **Расположение параметра.**
④ **Является ли параметр обязательным.**
⑤ **Описание структуры данных параметра.**
⑥ **Тип параметра (строка).**

Мы указываем, что параметр находится в (in) запросе (query), но не является обязательным, и что его структура данных описана в схеме (schema). Данная схема просто указывает на то, что тип этого параметра – строка. Мы также предоставляем дополнительную информацию в свойстве description, чтобы сообщить, что ее значением может быть A product's name, reference, or partial description.

Свойство parameters представляет собой список или массив. В YAML каждый элемент списка или массива начинается с дефиса (-). Чтобы описать параметр, нам нужно как минимум три свойства: name, in и schema. Описание этого параметра также содержит два необязательных свойства: required и description.

name – это имя, которое будет отображаться в пути (/products?free-query={free query}). Свойство in указывает местоположение параметра. Здесь это параметр query, поэтому он расположен после символа ?.

Свойство required, которое указывает на то, должен ли предоставляться параметр, не является обязательным. Если значение для него не задано, его значение по умолчанию – false. Это указывает на то, что параметр является необязательным. Вы не должны устанавливать для него значение, только если вам не нужно определять параметр как required. Но, хотя этот параметр и является необязательным, всегда лучше явно указать required: false. Таким образом, вы уверены, что изучили, является ли каждый параметр обязательным или нет.

**СОВЕТ.** Даже если свойство description не является обязательным, я рекомендую всегда указывать его. Одного имени параметра чаще всего недостаточно для описания того, что вы ожидаете.

Структура данных параметра, описанная в свойстве schema, является схемой JSON. Как упоминалось ранее, JSON Schema используется в документе OAS для описания данных API – от простых строковых параме-

тров запроса до более сложных структур, применяемых в качестве параметров тела и ответов. Используя JSON Schema, давайте посмотрим, как описать товар, например тот, что мы спроектировали в разделе 3.3.1.

## 4.3.2 Описание данных с помощью JSON Schema

Начнем с очень простой версии товара, как показано на рис. 4.13. Он состоит из `reference`, `name` и `price`. Свойства `reference` и `name` имеют тип «строка», а `price` – это число.

Описание товара

| Имя | Тип |
| --- | --- |
| reference | string |
| name | string |
| price | number |

Пример товара

```
{
  "reference": "ISBN-9781617295102",
  "name": "The Design of Web APIs",
  "price": 44.99
}
```

**Рис. 4.13. Базовое описание товара**

Ранее, чтобы описать простой строковый параметр запроса с помощью JSON Schema, мы использовали это: `type:string`. Теперь, чтобы описать такой объект-товар, мы должны взять тип object и перечислить его свойства. Каждое свойство идентифицируется по имени и типу, как показано в этом листинге.

**Листинг 4.8. Описание базового товара с помощью JSON Schema**

```
type: object ①
properties: ②
  reference: ③
    type: string ④
  name: ③
    type: string ④
  price: ③
    type: number ④
```

① **Эта схема описывает объект.**
② **Здесь содержатся свойства.**
③ **Имя свойства.**
④ **Тип свойства.**

Но когда мы обсуждали проектирование данных API в предыдущей главе, вы узнали, что также нужно определить, какие требуются свойства. Поскольку свойства `reference`, `name` и `price` являются обязательными, для примера добавим дополнительное свойство description (см. рис. 4.14).

Описание товара

| Имя | Тип | Обязательное свойство или нет |
|---|---|---|
| reference | string | да |
| name | string | да |
| price | number | да |
| description | string | нет |

Пример товара

```
{
  "reference": "ISBN-9781617295102",
  "name": "The Design of Web APIs",
  "price": 44.99,
  "description": "A book about web API design"
}
```

**Рис. 4.14. Описание товара с необходимыми флагами**

Теперь товар состоит из обязательных свойств `reference`, `name` и `price` и необязательного свойства `description`. Чтобы указать это в соответствующей JSON-схеме, мы добавляем записи для `reference`, `name` и `price` в список обязательных свойств объекта, как показано в этом листинге.

**Листинг 4.9. Обязательные и необязательные свойства товара**

```
type: object
required: ①
  - reference
  - name
  - price
properties:
  reference: ②
    type: string
  name: ②
    type: string
  price: ②
    type: number
  description: ③
    type: string
```

① **Список обязательных свойств.**
② **Обязательные свойства.**
③ **Необязательное свойство.**

Схема JSON позволяет нам указать, какие свойства требуются в объекте с помощью списка `required`. Любое свойство, имя которого включено в этот список, является обязательным. Любое свойство, имя которого отсутствует в этом списке, считается необязательным. В данном случае `description` – единственное необязательное свойство.

Наша схема становится довольно точной, но когда мы проектировали ресурс товара, то обнаружили, что нам иногда нужно было добавлять некоторые описания, потому что имен свойств было недостаточно для объяснения их природы. Давайте добавим описание к объекту, чтобы показать, что он описывает `A product`. Мы также можем добавить описание, чтобы объяснить, что такое `reference`, и даже добавить пример,

чтобы показать, как выглядит ссылка на товар. Это показано в приведенном ниже листинге.

**Листинг 4.10. Документирование схемы JSON**

```
type: object
description: A product ①
required:
  - reference
  - name
  - price
properties:
  reference:
    type: string
    description: Product's unique identifier ②
    example: ISBN-9781617295102 ③
  name:
    type: string
    example: The Design of Web APIs ③
  price:
    type: number
    example: 44.99 ③
  description:
    type: string
    example: A book about API design ③
```

① **Описание объекта.**
② **Описание свойства.**
③ **Пример значения свойства.**

Как и OAS, JSON Schema поставляется с полезными функциями документирования. Можно описать объект и все его свойства, и для каждого свойства может быть предоставлено примерное значение.

Но мы по-прежнему что-то упускаем. Ресурс товара, который мы разработали в предыдущей главе, имел не только буквальные свойства (строки или числа), как показано на рис. 4.15.

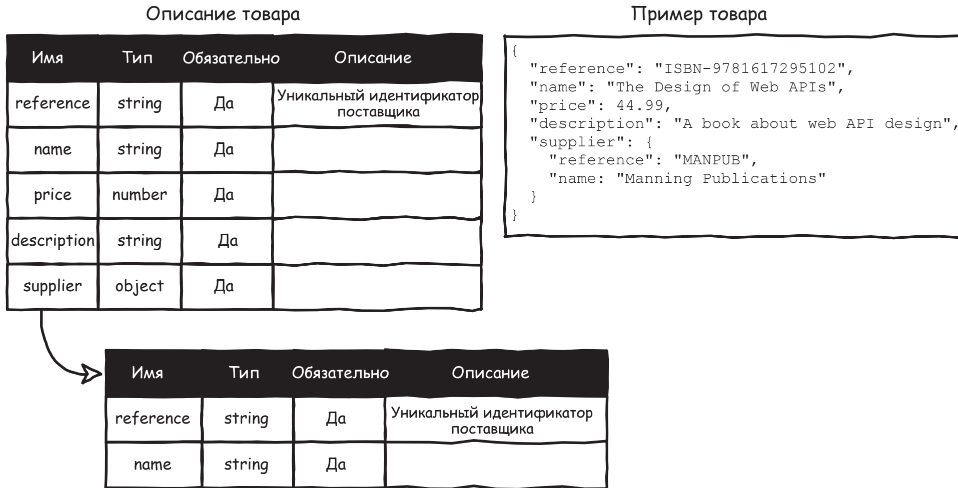



**Рис. 4.15.** Товар с описанием поставщика

В действительности, у него также было комплексное свойство `supplier`, которое является обязательным. Свойство `supplier` определяется обязательными `reference` и `name`. В приведенном ниже листинге показана обновленная схема JSON.

**Листинг 4.11.** Описание комплексного свойства с помощью JSON Schema

```
type: object
description: A product
required:
   - reference
   - name
   - price
   - supplier ①
properties:
   reference:
      type: string
      description: Product's unique identifier
      example: ISBN-9781617295102
name:
   type: string
   example: The Design of Web APIs
price:
   type: number
   example: 44.99
description:
   type: string
   example: A book about API design
supplier: ②
   type: object
```

```
    description: Product's supplier
    required: ③
      - reference
      - name
    properties: ④
      reference:
        type: string
        description: Supplier's unique identifier
        example: MANPUB
      name:
        type: string
        example: Manning Publications
```

① **Необходимо указать поставщика.**
② **Свойство объекта supplier.**
③ **Необходимые свойства.**
④ **Описание свойств.**

Чтобы добавить свойство `supplier` в JSON-схему товара, включаем его в список `properties` и устанавливаем для его типа значение `object`. Мы предоставляем описание для свойства и список обязательных свойств (`reference` и `name`); затем описываем эти свойства.

Для свойства `reference` мы предоставляем `type`, `description` и `example`. Для свойства name устанавливаем значения только для `type` и `example`. Наконец, добавляем свойство `supplier` в список товара `required`.

Как видите, описывать данные с использованием JSON Schema очень просто; вы можете описать любую структуру данных, используя этот формат. К сожалению, в этой книге не рассказывается о всех его возможностях. Чтобы узнать больше, я рекомендую прочитать описание Объекта схемы в OAS (https://github.com/OAI/OpenAPI-Specification/tree/master/versions), а затем спецификацию JSON Schema (http://json-schema.org/specification.html). Теперь, когда вы знаете, как описывать структуры данных с помощью JSON Schema, давайте опишем ответ на поиск товаров.

### 4.3.3 Описание ответов

Когда пользователи ищут товары, они должны получать товары, соответствующие предоставленному бесплатному запросу (рис. 4.16). Соответствующий API-запрос `GET /products?free-query={free query}` возвращает HTTP-ответ `200 OK`, тело которого будет содержать массив товаров.

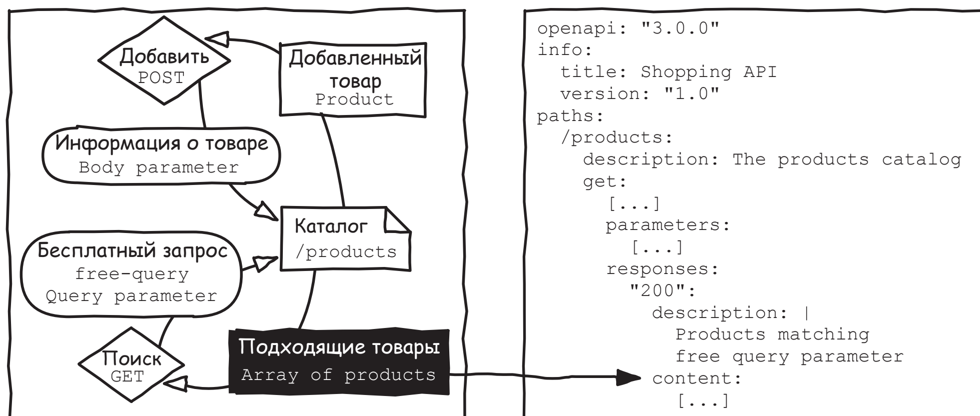



**Рис. 4.16. Описание ответа на поиск товаров**

В документе OAS данные, возвращаемые операцией в теле HTTP-ответа, определяются в свойстве `content`, как показано в листинге 4.12. При описании содержимого ответа вы должны указать медиа тип документа, содержащегося в теле ответа. Медиа тип указан в HTTP-ответе. На данный момент, как было сказано ранее, мы будем считать само собой разумеющимся, что наш API возвращает документы в формате JSON, поэтому мы указываем в ответе `application/json`. ( Поговорим об этой функции HTTP позже, в главе 6.)

**Листинг 4.12. Описание данных ответа**

```
openapi: "3.0.0"
info:
  title: Shopping API
  version: "1.0"
paths:
  /products:
    get:
    summary: Search for products
    description: | Search using a free query (query parameter)
    parameters:
      [...]
    responses:
      "200":
        description: Products matching free query
        content: ①
          application/json: ②
            schema: ③
              [...]
```

① **Определение тела ответа.**

② **Медиа тип тела ответа.**

③ **JSON-схема тела ответа.**

Как только это будет сделано, мы можем описать схему возвращаемого документа JSON, используя JSON Schema, как показано в листинге 4.13. Действие `GET /products` возвращает массив товаров. Мы уже знаем, как описывать один товар, используя JSON Schema, но как описать массив товаров? Как видно в приведенном ниже листинге, массив описывается с использованием типа `array`. Свойство `items` содержит схему элементов массива. Этот массив имеет несколько товаров; вы должны узнать схему JSON, созданную ранее.

**Листинг 4.13. Описание массива товаров**

```
responses:
  "200":
    description: Products matching free query
    content:
      application/json: ①
        schema: ②
          type: array ③
          description: Array of products
          items: ④
            type: object
            description: A product
            required:
              - reference
              - name
              - price
              - supplier
          properties:
            reference:
              description: Unique ID identifying a product
              type: string
            name:
              type: string
            price:
              description: Price in USD
              type: number
            description:
              type: string
            supplier:
              type: object
              description: Product's supplier
              required:
                - reference
                - name
              properties:
                reference:
                  type: string
```

```
              name:
                type: string
```

① Медиа тип тела ответа.
② JSON-схема тела ответа.
③ Тип ответа – массив.
④ Схема элементов массива.

В следующем листинге содержится пример документа в формате JSON, возвращаемого в теле ответа, в соответствии со схемой JSON.

**Листинг 4.14. Пример массива товаров в формате JSON**

```
[
  {
    "reference": "123-456",
    "name": "a product",
    "price": 9.99,
    "supplier": {
      "reference": "S789",
      "name": "a supplier"
    }
  },
  {
    "reference": "234-567",
    "name": "another product",
    "price": 19.99,
    "supplier": {
      "reference": "S456",
      "name": "another supplier"
    }
  }
]
```

Как видите, описание данных ответа – процесс довольно простой. И знаете что? Описывать параметры тела так же просто.

### 4.3.4 Описание параметров тела

Давайте посмотрим на действие «Добавить товар». Чтобы добавить товар в каталог, пользователь API должен предоставить какую-то информацию о товаре в теле своего запроса, как показано на рис. 4.17.

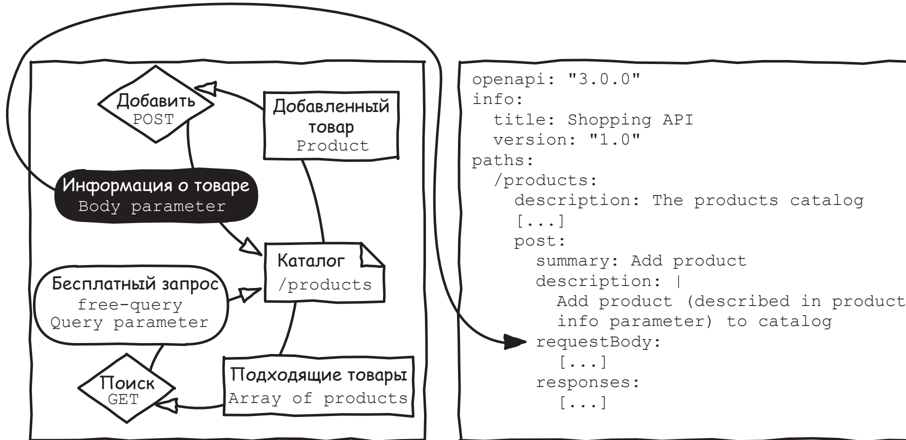



**Рис. 4.17. Описание параметра тела запроса «Добавить товар»**

Описание параметра тела для действия «Добавить товар» выполняется почти так же, как и описание ответа при поиске товаров, как показано в этом листинге.

**Листинг 4.15. Описание параметра тела действия**

```
openapi: "3.0.0"
   info:
   title: Shopping API
version: "1.0"
paths:
   /products:
   description: The products catalog
   [...]
   post:
      summary: Add product
      description: Add product to catalog
      requestBody: ①
         description: Product's information ②
         application/json: ③
            schema: ④
               [...]
      responses:
         "200":
            description: Product added to catalog
```

① **Определение параметра тела.**
② **Описание параметра тела.**
③ **Медиа тип параметра тела.**
④ **Схема параметра тела.**

Параметр тела HTTP-запроса описан в его свойстве `requestBody`. Как и тело ответа, параметр `body` имеет медиа тип (`application/json`), а его

содержимое описывается схемой JSON. Полное описание этого параметра приведено в приведенном ниже листинге.

**Листинг 4.16. Полное описание параметра тела**

```
requestBody:
  description: Product's information
  content:
    application/json:
      schema: ①
        required:
          - name
          - price
          - supplierReference
        properties:
          name:
            type: string
          price:
            type: number
          description:
            type: string
          supplierReference:
            type: string
```

① **Схема параметра тела.**

Схема параметра тела (или схема тела запроса) описывается, как и любые другие данные в OAS, с использованием схемы JSON. Как и предполагалось ранее, обязательной информацией, необходимой для добавления товара в каталог, является `name`, `price` и `supplierReference`. Свойство `description` – необязательно. В приведенном ниже листинге показано, как может выглядеть документ JSON.

**Листинг 4.17. Пример информации о товаре в формате JSON**

```
{
  "name": "a product",
  "price": 9.99,
  "supplierReference": "S789"
}
```

Видите? Это было просто, как и ожидалось. Почему? Потому что описание тела запроса и ответа выполняется одинаково. Предоставление общего способа делать разные вещи является основным принципом проектирования. Данный подход можно использовать при создании чего-угодно, от двери до формата описания API, чтобы сделать это удобным для пользователя. Мы рассмотрим это с точки зрения проектирования API позже, в главе 6.

Нам нужно только описать ответ на добавление товара, чтобы завершить описание ресурса каталога. Мы узнали все, что нам нужно для это-

го. Мы знаем, как описать ответ действия и его данные. И к счастью, при поиске и добавлении товаров возвращаются данные того же типа, что и товар, поэтому у нас уже есть схема JSON, описывающая данные ответа. Но если мы сделаем это, как научились, мы продублируем JSON-схему товара в документе OAS. Давайте посмотрим, как справиться с этим более эффективно.

## 4.4 Эффективное описание API с помощью OAS

Всегда полезно покопаться в документации формата описания API, чтобы узнать все его советы и рекомендации, как это делается при изучении языка программирования. Есть две основные вещи, которые вы должны знать при написании документов OAS:

- как повторно использовать компоненты, такие как схемы JSON, параметры или ответы;
- как эффективно определить параметры пути.

### 4.4.1 Повторное использование компонентов

И при поиске, и при добавлении товаров возвращаются ресурсы товара. Было бы жаль описывать одно и то же дважды. К счастью, OAS позволяет нам описывать повторно используемые компоненты, такие как схемы, параметры, ответы и многое другое, и использовать их, где это необходимо с помощью ссылки (рис. 4.18).

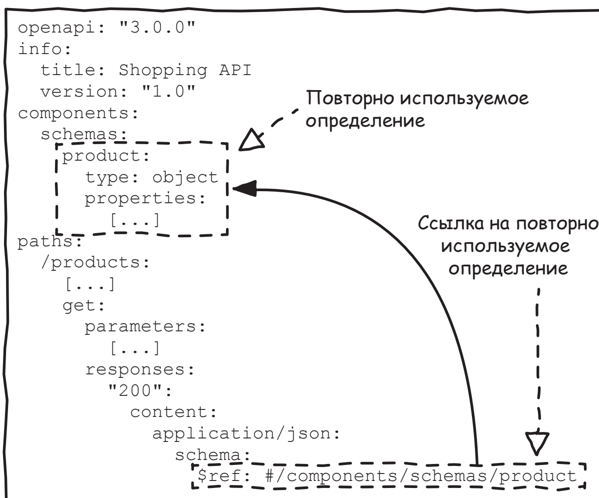



Рис. 4.18. Многократно используемые компоненты в документе OAS

Чтобы избежать двукратного описания JSON-схемы товара, нам нужно всего лишь объявить ее *схемой многократного использования*. Многократно используемые компоненты описаны в разделе `components`, который находится в корне документа OAS. В этом разделе повторно используемые схемы определены в `schemas`; имя каждой такой схемы опреде-

ляется как свойство schemas. Это свойство содержит повторно используемый компонент, схему JSON. В приведенном ниже листинге товар содержит JSON-схему, которую мы создали ранее.

**Листинг 4.18. Объявление схемы многократного использования**

```
openapi: "3.0.0"
[...]
components: ①
  schemas: ②
    product: ③
      type: object ④
      description: A product
      required:
        - reference
        - name
        - price
        - supplier
      properties:
        reference:
          description: |
            Unique ID identifying
            a product
          type: string
        name:
          type: string
        price:
          description: Price in USD
          type: number
        description:
          type: string
      supplier:
          type: object
          description: Product's supplier
          required:
            - reference
            - name
          properties:
            reference:
              type: string
            name:
              type: string
```

① **Компоненты многократного использования.**
② **Схемы многократного использования.**
③ **Имя схемы многократного использования.**
④ **Схема JSON.**

Теперь вместо переопределения JSON-схемы товара мы можем использовать ссылку JSON для доступа к этой предопределенной схеме, когда она нам понадобится. *Ссылка JSON* – это свойство, имя которого – `$ref`, а содержимое – URL-адрес. Этот URL-адрес может указывать на любой компонент внутри документа или даже в других документах. Поскольку мы только ссылаемся на локальные компоненты, мы используем локальный URL-адрес, содержащий только фрагмент, описывающий путь к нужному элементу, как показано на рис. 4.19. Здесь `product` находится в `schemas`, которые расположены в компонентах в корне документа.

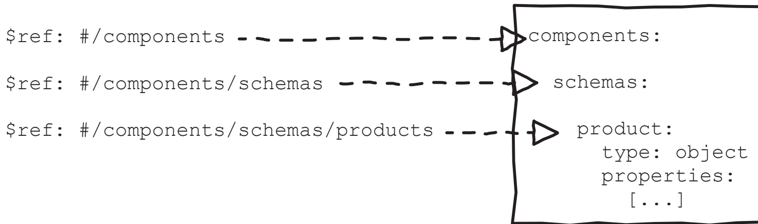



**Рис. 4.19. JSON-ссылки на локальные компоненты**

В приведенном ниже листинге показано, как можно использовать ссылку для ответа `POST /products`.

**Листинг 4.19. Использование предопределенного компонента с ссылкой**

```
post:
  summary: Add product
  description: Add product to catalog
  [...]
  responses:
    "200":
      description: Product added to catalog
      content:
        application/json:
          schema: ①
          $ref: "#/components/schemas/product" ②
```

① **Схема ответа.**

② **Ссылка на предопределенную схему.**

Когда пользователи добавляют товар в каталог, они получают взамен созданный товар. Таким образом, при определении схемы ответа вместо повторного описания JSON-схемы товара мы можем просто использовать свойство `$ref`, значение которого – это ссылка на предопределенную схему. Можно сделать то же самое для поиска товара, когда возвращается массив товаров, как показано в этом листинге.

**Листинг 4.20. Использование предопределенного компонента в массиве**

```
get:
  summary: Search for products
```

```
description: |
  Search using a free query (query parameter)
parameters:
  [...]
responses:
  "200":
    description: Products matching free query
    content:
      application/json:
        schema: ①
          type: array ②
          items: ③
            $ref: "#/components/schemas/product" ④
```

① **Схема ответа.**
② **Массив.**
③ **Схема элементов массива.**
④ **Ссылка на предопределенную схему.**

Предопределенная нами схема описывает только один товар, а не массив товаров, поэтому здесь мы используем предопределенную схему для описания схемы массива `items`. Чтобы сделать это, мы, как и раньше, просто заменим схему ссылкой (`$ref`) на предопределенную схему. Это означает, что мы можем сочетать встроенные и предопределенные определения. Обратите внимание, что мы также можем использовать несколько предопределенных определений, когда это необходимо.

Что касается ресурса каталога, идентифицированного путем `/products` и его двумя действиями, `get` и `post`, то мы закончили. Эти элементы полностью и эффективно описаны благодаря OAS и JSON Schema. И еще одна вещь, которую следует изучить, чтобы можно было полностью описать базовый REST API, – как описать ресурс с помощью переменного пути.

### 4.4.2 Описание параметров пути

Ресурс-товар, который можно удалить, обновить или заменить, идентифицируется переменным путем (рис. 4.20). Обратите внимание, что действия, обозначенные глаголами *get*, *update* и *replace*, возвращают один и тот же товар, а также что действия *update* и *replace* используют один и тот же параметр. Также заметьте, что путь `/products/{productId}` содержит переменную `productId`, которая называется *параметром пути*.

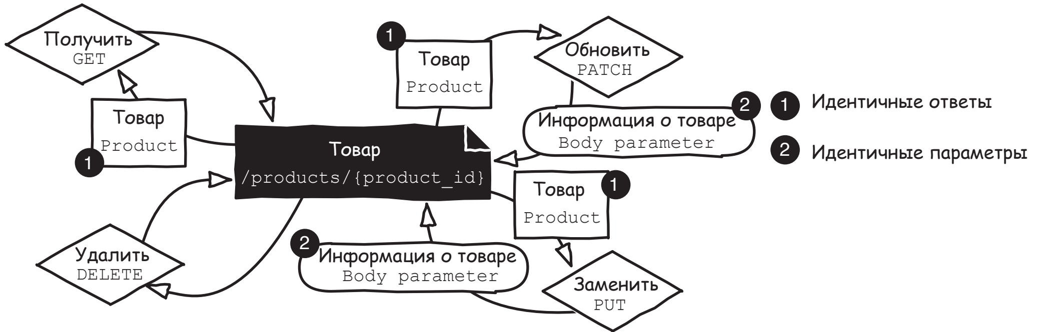



**Рис. 4.20.** Ресурс товар и его действия

Вы уже узнали, как определять параметры на уровне действий с помощью запроса `GET /products?free-query={free query}`, поэтому давайте сделаем это снова с помощью `DELETE /products/{productId}`.

В приведенном ниже листинге показано, как можно определить это в нашем документе OAS для действия удаления.

**Листинг 4.21. Удаление товара**

```
paths:
  /products:
  [...]
  /products/{productId}: ①
    description: A product
    delete: ②
      summary: Delete a product
      parameters: ③
        - name: productid ④
          in: path ⑤
          required: true ⑥
          description: Product's reference
          schema:
            type: string
```

① **Путь ресурса товара с параметром.**
② **Действие "Удалить товар".**
③ **Параметры действия "Удалить товар".**
④ **Имя параметра пути.**
⑤ **Параметр находится в пути.**
⑥ **Параметр является обязательным.**

Сначала мы добавляем новый путь `/products/{productId}`, чтобы определить ресурс товара. Как видно, параметр пути обозначен фигурными скобками (`{productId}`) в `paths`. Затем определяем этот параметр пути в списке действия удаления `parameters`. Он определяется почти так же, как и любой другой параметр, который идет в разделе `parameters`: нам нужно установить значения для `name`, `location` и `schema`. `name` должно соответствовать имени внутри фигурных скобок в пути, поэтому мы

устанавливаем для него значение productId. Местоположение (in), очевидно, является путем, а тип этого параметра, определенный в его схеме, – строкой.

Мы почти закончили, но есть еще одна вещь, которую не стоит забывать: поскольку это параметр пути, мы также должны сделать этот параметр обязательным, установив для required значение true. Если этого не сделать, парсер выкинет ошибку.

Это не сильно отличается от определения параметра запроса. А что, если мы захотим описать обновление и замену товара? Можно было бы описать параметр пути таким же образом, но это означало бы дублирование этого описания в каждом новом действии. Как сделать это более эффективно?

Ранее мы обнаружили раздел components документа OAS. Этот раздел позволяет нам определить компоненты многократного использования, такие как схемы и ответы, и здесь мы также можем описывать параметры многократного использования. В приведенном ниже листинге показано, как это делается.

**Листинг 4.22. Описание многократно используемого параметра**

```
components: ①
  parameters: ②
    productId: ③
    name: productId
    in: path
    required: true
    description: Product's reference
      schema:
        type: string
```

① **Компоненты многократного использования.**

② **Параметры многократного использования.**

③ **Имя параметра многократного использования.**

Чтобы определить параметр многократного использования, мы делаем то же самое, что и со схемой многократного использования. В разделе components в корне документа OAS каждый повторно используемый параметр определяется как свойство parameters и идентифицируется по имени. Свойство productId содержит определение параметра пути productId, как мы его определили для DELETE /products/{productId}.

Довольно просто, не правда ли? Как и в схеме JSON, мы используем этот предопределенный параметр со ссылкой JSON, как показано в этом листинге.

**Листинг 4.23. Использование предопределенного параметра**

```
components:
  parameters:
    productId: ①
```

```
      [...]
paths:
  /products:
  [...]
  /products/{productId}: ②
  delete:
    parameters:
      - $ref: #/components/parameters/productid ③
    [...]
  put:
    parameters:
      - $ref: #/components/parameters/productid ③
    [...]
  patch:
    parameters:
      - $ref: #/components/parameters/productid ③p
    [...]
```

① **Определение параметра пути.**
② **Путь ресурса товара с параметром.**
③ **Ссылка на предопределенный параметр.**

Вместо того чтобы определять один и тот же параметр трижды, мы просто применяем свойство `$ref`, указывающее на уникальное и повторно используемое определение `productId`.

Так гораздо лучше! Параметр `productId` определяется один раз и используется в трех разных местах. Но знаете что? Можно сделать еще лучше. Строго говоря, параметр productId не является параметром действия; это параметр ресурса.

В документе OAS параметры могут определяться не только на уровне действий, но и на уровне ресурсов, опять же в разделе `parameters`. Структура этого раздела точно такая же, как и на уровне действий. Все параметры, определенные на уровне ресурса, применимы ко всем действиям на ресурсе. Поэтому, как показано в приведенном ниже листинге, чтобы еще больше упростить наш документ, можно только определить параметр пути `productId` в разделе `parameters` пути `/products/{productId}`.

**Листинг 4.24. Параметры уровня ресурса**

```
components:
  parameters:
    productId: ①
      [...]
paths:
  /products:
  [...]
  /products/{productId}:
    parameters: ②
      - $ref: #/components/parameters/productId ③
```

```
    delete: ④
      [...]
    put: ④
      [...]
    patch: ④
      [...]
```

① **Определение параметра пути.**
② **Параметры уровня ресурса.**
③ **Ссылка на предопределенный параметр.**
④ **Определений параметров пути больше нета.**

Поздравляю! После того, что вы теперь узнали о OAS, вы сможете завершить описания действий ресурса товара. Но, что еще более важно, теперь вы способны создать формальное описание любого базового REST API с использованием OAS и поделиться им со всеми, кто участвует в вашем проекте. Не стесняйтесь покопаться в документации OAS (https://github.com/OAI/OpenAPI-Specification/tree/master/versions), используйте мою карту OpenAPI (https://openapi-map.apihandyman.io) и экспериментируйте, чтобы обнаружить другие возможности.

Этой главой я завершаю первую часть этой книги. Вы приобрели базовый набор навыков проектирования API, и теперь знаете:

- что такое API на самом деле;
- как идентифицировать свои цели с точки зрения потребителя;
- как перенести их в программируемое представление;
- как формально описать это программируемое представление с помощью OAS.

В следующей части мы усовершенствуем эти навыки, чтобы вы могли создавать API-интерфейсы, которые любой может с легкостью использовать, даже не задумываясь об этом. В следующей главе мы подробно рассмотрим удобство использования API, изучив, как разрабатывать простые API.

## Резюме

- Формат описания API – это простой и структурированный способ описания и совместного использования программного интерфейса.
- Документ описания API – это машиночитаемый документ, который можно использовать различными способами, в том числе для создания справочной документации по API.
- Формат описания API используется только при разработке программируемого представления и данных API, но не раньше.
- Всегда используйте возможности документации формата описания API. Подробно изучите документацию формата описания API, чтобы иметь возможность эффективно его использовать и в особенности определять повторно используемые компоненты, где это возможно.

# *Часть 2*

# *Проектирование практичного API*

Теперь, закончив первую часть этой книги, вы можете идентифицировать реальные цели API и представить их как программный интерфейс, учитывая при этом точку зрения потребителя и избегая точки зрения поставщика. Это прочная основа для проектирования API, который делает свою работу, – но люди ожидают от API большего, чем простого *выполнения работы*. API ничего не стоит, если он непрактичен. Чем более он практичен, тем меньше усилий требуется для работы с ним и тем больше людей может использовать его. Им даже может понравиться использовать его. *Юзабилити*, или удобство использования, – это то, что отличает потрясающие API от посредственных или сносных.

Удобство использования жизненно важно для любого API, как и для любого повседневного объекта. Помните кухонный радар 3000? У него был ужасный интерфейс, потому что он просто предоставлял доступ к внутренней работе. К сожалению, такой скверный дизайн может иметь место при создании API, даже если избегать опасной точки зрения по-

ставщика! К счастью, основным принципам юзабилити можно научиться, наблюдая за повседневными вещами.

Когда люди используют повседневные предметы, они хотят достичь своих целей, не обременяя себя обильным, но нечетким набором функций. И им нравится думать, что они умны, потому что они могут узнать все об объекте самостоятельно. То же самое и в случае с API.

Потребители не хотят думать, когда используют API; им нужны API с простым дизайном, который позволяет мгновенно достигать своих целей, не тратя время на понимание данных, функций или отзывов об ошибках. Они также хотят иметь четкие и организованные наборы целей, а не ошеломляющий, гигантский, пестрый API.

И самое главное – они хотят чувствовать себя как дома, когда используют API; испытывать чувство дежавю, потому что API рассказывает им все, а его дизайн соответствует стандартам или общепринятым практикам.

# 5 Проектирование простого API

**В этой главе мы рассмотрим:**

- создание простых представлений концепций;
- идентификацию релевантных сообщений об ошибках и успехах;
- проектирование эффективных потоков использования;
- как описывать REST API с помощью спецификации OpenAPI (OAS).

Теперь, когда вы научились проектировать API, которые фактически позволяют потребителям добиваться своих целей, у вас есть прочная основа для проектирования API. К сожалению, полагаться только на основы не означает, что потребители действительно смогут использовать «API, которые делают свою работу». Помните UDRC 1138, показанный на рис. 5.1? Разработать ужасный интерфейс, который делает свою работу – возможно.

**Рис. 5.1. Ужасный интерфейс, который делает свою работу**

Что вы делаете, когда сталкиваетесь с незнакомым предметом обихода? Вы наблюдаете за ним. Вы анализируете его форму, надписи, значки, кнопки или другие элементы управления, чтобы получить представ-

ление о его назначении, текущем состоянии и о том, как с ним работать. Чтобы достичь своей цели, используя этот объект, вам может потребоваться объединить различные взаимодействия, предоставляя входные данные и получая отклик. Делая все это, вы не хотите неопределенности; все должно быть кристально ясно. Не хочется терять время, поэтому все должно идти быстро и эффективно. Вы хотите, чтобы ваш опыт использования любого повседневного предмета был максимально простым. Вы определенно не желаете сталкиваться с еще одним интерфейсом, подобным UDRC 1138. Это основа *юзабилити*. То же самое касается и API.

Люди ожидают, что API будут практичными. Они ожидают простых представлений, простых взаимодействий и простого потока. Мы раскроем некоторые фундаментальные принципы юзабилити, наблюдая за повседневными вещами в различных ситуациях, а затем перенесем эти принципы в проектирование API. Сейчас и на протяжении оставшейся части книги мы будем работать над воображаемым розничным банковским API, предоставленным фиктивной банковской компанией. Этот API может использоваться, например, приложением мобильного банкинга для получения информации о текущих счетах, такой как баланс и транзакции, и перевода денег с одного счета на другой. Давайте начнем с того, что научимся создавать представления, отвечающие этим требованиям.

## 5.1 Проектирование простых представлений

То, как проектировщик решает представлять концепции и информацию, может значительно улучшить или ухудшить удобство использования. Если избегать точки зрения поставщика и сосредоточиться на потребителе, как вы узнали в главе 2, это будет очевидным первым шагом на пути к юзабилити, но нам также необходимо позаботиться и о некоторых других аспектах, чтобы гарантировать, что мы создаем простые представления. Давайте поработаем над обычным будильником, чтобы найти эти аспекты.

На рис. 5.2 показано, как можно изменить внешний вид будильника; как можно исправить используемые представления, чтобы сделать его наименее пригодным для использования.

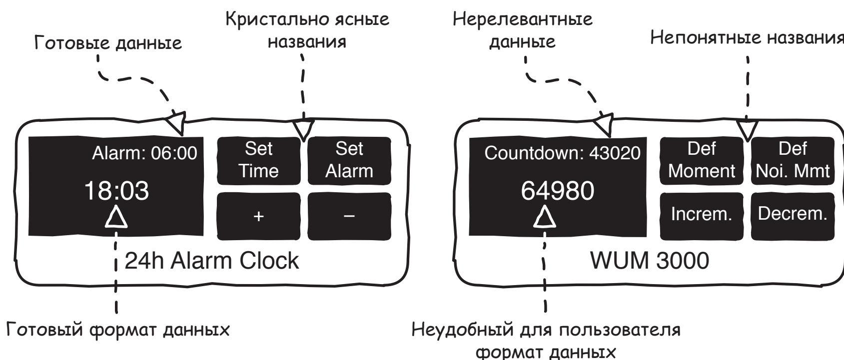



Рис. 5.2. Превращение будильника в устройство, которое не так удобно использовать

Во-первых, мы можем использовать больше непонятных надписей. 24-часовой будильник превращается в WUM 3000 (Wake Up Machine 3000). Кнопки **Set Time**, **Set Alarm** и **+** и **–** заменяются на **Def Moment** (Определить момент), **Def Noi. Mmt** (Определить момент шума), **Increm**. (Увеличение) и **Decrem**. (Понижение). Мы также можем использовать менее удобный для пользователя формат текущего времени и заменить его количеством секунд, прошедших с полуночи, поэтому 18:03 превращается в 64 980. И наконец, можно заменить время звонка тесно связанной, но менее полезной информацией: например, обратным отсчетом времени звонка в секундах. Если текущее время – 18:03, то время звонка 06:00 превращается в 43 020 (и идет отсчет).

Как и 24-часовой будильник, WUM 3000 не открывает доступ к сложности внутреннего устройства. Оба сделаны, для того чтобы показывать время и, что более важно, они издают какой-то ужасный шум в заданный момент времени. Но используемые представления немного отличаются, и это влияет на удобство применения. 24-часовой будильник можно использовать, потому что человек может понять, что это за устройство, каково его текущее состояние и для чего оно, просто читая надписи, кнопки и экран дисплея. С другой стороны, WUM 3000 гораздо менее пригоден для использования, потому что эти элементы довольно непонятны и сбивают с толку.

Точно так же и с API. Выбор, который вы делаете в отношении имен, форматов данных и данных, может значительно улучшить или навредить удобству использования API. Вы уже знаете, как спроектировать эти вещи; просто нужно подумать о том, имеют ли выбранные представления смысл и легко ли они понятны потребителю, фокусируясь на точке зрения потребителя и проектируя с учетом потребностей пользователя. Давайте рассмотрим эти темы более внимательно, чтобы полностью понять, как создавать простые представления.

### *5.1.1 Выбор кристально ясных имен*

Невозможно определить, что такое WUM 3000 и как его использовать, основываясь только на его названии или его кнопках **Def Moment** и **Def Noi. Mmt**, в то время как 24-часовой будильник – это, очевидно… будильник. Кодовые имена, неуклюжие слова и загадочные сокращения могут сделать обыденный предмет совершенно непонятным. То же самое относится и к API.

Когда вы анализируете потребности с помощью таблицы целей API, вы должны назвать входные и выходные данные. Эти данные должны быть представлены в виде ресурсов, ответов или параметров, у каждого из которых есть имена. Эти элементы могут содержать свойства, у которых также имеются имена. Представляя их с помощью такого инструмента, как спецификация OpenAPI (OAS), вам, возможно, придется выбирать имена для JSON-схем многократного использования. В зависимости от выбранного стиля API цели могут быть представлены в виде функций, у которых также есть имена. Повсюду имена. Итак, как же их выбирать?

В разделе 2.2.2 вы познакомились с точками зрения потребителя и поставщика. Мы уже знаем, что мы должны выбрать имена, которые что-то значат для потребителей. Но даже зная это, нужно быть осторожными при создании этих имен.

Допустим, текущие счета банковской компании оснащены дополнительной функцией защиты от овердрафта. Если она активна, банк не будет взимать комиссионные в случае, когда снимаемая клиентом сумма превышает доступный остаток. Было бы интересно узнать, активна ли эта опция или нет при получении информации о банковском счете с помощью банковского API, предоставленного этой компанией. На рис. 5.3 показано, как это можно представить в API.

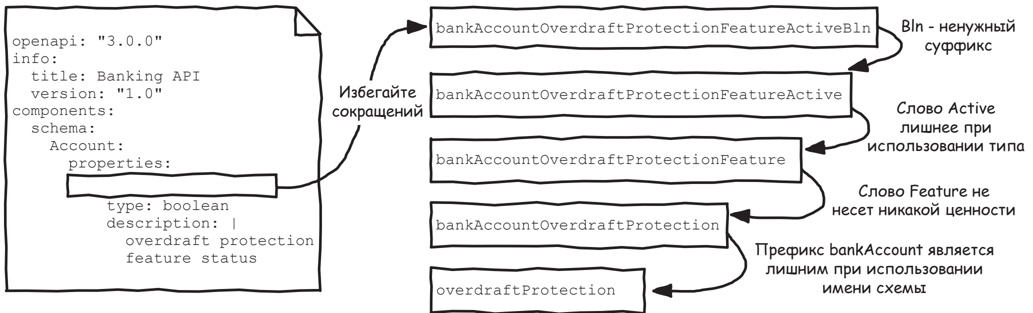



**Рис. 5.3. Выбор имени свойства**

Первая идея – использовать булево свойство с именем `bkAccOverProtFtActBln`, которое имеет значение `true`, когда функция активирована, и `false` – если нет. Но хотя использование логического типа данных полностью имеет смысл, имя `bkAccOverProtFtActBln` не является, по меньшей мере, удобным для пользователя. Использование аббревиатур, как правило, не очень хорошая идея, потому что это затрудняет их понимание. (Обратите внимание, что такие сокращения, как *max* или *min*, являются приемлемыми, поскольку они широко используются; мы поговорим об этом подробнее в разделе 6.1.3.)

Если уменьшить количество сокращений, это имя превращается в `bankAccountOverdraft-ProtectionFeatureActiveBln`; это описательное и читабельное имя. Но, хотя оно и понятно, такое имя – ужасно длинное. Давайте посмотрим, как найти более подходящую и вместе с тем легко воспринимаемую альтернативу.

Суффикс Bln указывает на то, что это свойство является логическим типом данных. Некоторые соглашения о кодировании могут способствовать использованию таких префиксов или суффиксов, как `bln`, `dto`, `sz`, `o`, `m_` и т. д., для объяснения технической природы свойства, класса или структуры. Поскольку вы знаете о точке зрения поставщика, вы, наверное, уже догадались, что раскрытие внутренних соглашений по кодированию таким образом может быть неуместным. Но даже если мы отбросим точку зрения поставщика, будут ли такие технические детали иметь значение для потребителей? Вовсе нет.

Потребители будут иметь доступ к документации API, в которой это свойство описывается как логический тип данных. И при тестировании API проектировщики увидят, что это свойство является логическим типом данных, потому что его значение равно `true` или `false`. Таким образом, мы можем сократить имя до `bankAccountOverdraftProtectionFeatureActive`.

Поскольку потребители API видят, что это свойство является логическим типом данных, мы также можем избавиться от суффикса `Active`. Это слово – лишнее и, следовательно, не имеет абсолютно никакого информативности. Таким образом, можно сократить имя до `bankAccountOverdraftProtectionFeature`.

Говоря об информативности, представляет ли слово *Feature* какой-либо интерес? Не особо.

С функциональной точки зрения оно просто свидетельствует о том, что это свойство или сервис банковского счета. Это можно объяснить в документации свойства. Таким образом, мы также можем избавиться и от этого слова и сократить имя свойства до `bankAccountOverdraftProtection`.

И наконец, это свойство принадлежит *банковскому счету*, поэтому нет необходимости указывать очевидное в его названии. Следовательно, свойство можно просто назвать `overdraftProtection`. Вместо семи слов у нас осталось два.

**СОВЕТ.** Я рекомендую вам попробовать использовать не более трех слов для создания имен (независимо от их назначения).

По сути, при создании этого названия мы использовали слова, которые понятны потребителям, и воспользовались окружающим контекстом, чтобы найти короткое, но вместе с тем понятное имя. Мы также избежали сокращений и раскрытия внутренних кодовых соглашений. Это все, что вам нужно сделать, если вы хотите найти кристально чистые имена для ресурсов, параметров, свойств, схем JSON или чего-то еще, что требует имени.

Некоторые имена, которые мы выбираем при проектировании API, предназначены для идентификации данных. И, как мы видели в случае со свойством `overdraftProtection`, выбор подходящих типов данных может значительно облегчить понимание.

### *5.1.2 Выбор простых в использовании типов данных и форматов*

WUM 3000 показал нам, что неадекватное представление данных может помешать юзабилити. Такое значение, как 64 980, явно не является временем, и даже если пользователи знают, что это так, им все равно придется выполнить какие-то вычисления, чтобы расшифровать его истинное значение: 18:03. Отсутствие контекста, неправильный тип или формат данных могут затруднить их понимание и использование.

Но все это касается повседневного предмета, используемого людьми. В случае с API данные просто обрабатываются программным обеспечением, использующим API, и он может отлично интерпретировать сложные

форматы. Он может даже преобразовать данные, прежде чем показывать их конечному пользователю, если это необходимо. Итак, почему же мы должны заботиться о типах данных и форматах при проектировании API?

Никогда не следует упускать из виду тот факт, что API является *пользовательским интерфейсом*. Проектировщики полагаются не только на имена, чтобы понимать и использовать API, но также и на его данные. Разработчики часто анализируют образцы запросов и ответов, вызывают API вручную или анализируют возвращаемые данные в своем коде с целью обучения, тестирования или отладки. Им также может понадобиться манипулировать конкретными значениями в своем коде. Чтобы сделать все это, они должны уметь понимать данные. Если API использует только сложные или непонятные форматы данных, такое упражнение будет довольно сложным. Так же, как имена, которые должны быть поняты с первого взгляда, смысл необработанных данных API всегда должен быть кристально понятным для проектировщиков.

**СОВЕТ.** Выбор подходящих форматов данных при проектировании API так же важен, как и выбор подходящих имен, чтобы обеспечить простое представление в вашем API.

Как видно из раздела 3.3.1, API обычно используют базовые переносимые типы данных, такие как строка, число или логический тип данных. Выбрать тип для свойства может быть относительно просто. Если мы продолжим проектирование концепции банковского счета, которую начали в предыдущем разделе, добавить название счета и баланс довольно просто. Действительно, название банковского счета, очевидно, должно быть строкой, а баланс – числом. Но в некоторых случаях нужно быть осторожными при выборе типов данных и форматов, чтобы убедиться, что то, что вы проектируете, понятно людям, как показано на рис. 5.4. На рисунке – примеры данных банковского счета в двух разных версиях: на левой стороне данные не так просты в использовании; на правой стороне – наоборот.

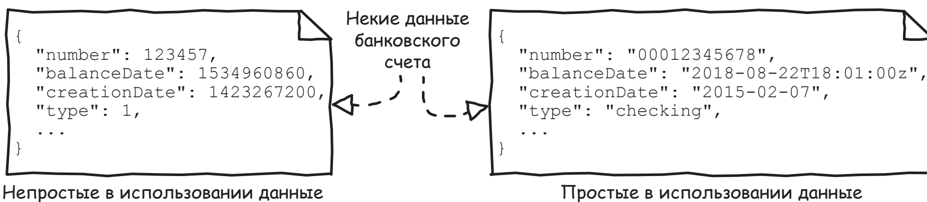



**Рис. 5.4.** Влияние типов данных и формата на удобство использования

Благодаря суффиксу `Date` и значению `1534960860` опытные разработчики должны понять, что `balanceDate` – это временная метка UNIX. Но смогут ли они расшифровать это значение, просто прочитав его? Возможно, нет. Его строковый аналог в формате ISO 8601, «`2018-08-22T18:01:00z`», гораздо более удобен для пользователя и понятен любому без какого-либо контекста и усилий (ну, почти без усилий, если вы знаете, что это «ГОД-МЕСЯЦ-ДЕНЬ», а не «ГОД-ДЕНЬ-МЕСЯЦ»).

То же самое относится и к свойству `creationDate`, значение которого слева равно `1423267200`. Но обратите внимание, что его значение в формате ISO 8601 показывает только дату без времени: «`2015-02-07`». Чтобы уменьшить риск неправильного обращения с часовым поясом, я рекомендую не указывать значение времени, когда это не является необходимостью.

Свойство `type` должно указывать, является ли счет расчетным или сберегательным. В то время как слева его значение – это непонятное число, 1, справа мы видим более явное строковое значение, `checking`. Использование числовых кодов – обычно плохая идея; они неудобны для людей, поэтому разработчикам придется постоянно обращаться к документации или изучать вашу номенклатуру, чтобы понять эти данные. Поэтому, если у вас есть такая возможность, лучше использовать типы данных или форматы, которые можно понять, просто прочитав их.

И наконец, коварный случай: если номер счета указан в виде числа (1234567), потребители должны быть осторожны и, возможно, сами должны добавить недостающие начальные нули. Строковое значение «`00012345678`» проще в использовании, и оно не повреждено.

Поэтому при выборе типов данных и формата нужно использовать то, что удобно для людей, и самое главное, всегда обеспечивать точное представление. При использовании сложного формата старайтесь предоставить достаточно информации. И, если возможно, постарайтесь, чтобы вас можно было понять без контекста.

Хорошо, теперь мы знаем, как выбирать имена и типы данных или форматы. Но юзабилити представления также зависит от того, какие данные мы решим использовать.

### *5.1.3 Выбор готовых к использованию данных*

WUM 3000 показал нам, что предоставление неверных данных может повлиять на удобство использования. Проектируя API, мы должны позаботиться о предоставлении релевантных и полезных данных, выходящих за рамки базовых данных, которые мы научились идентифицировать в разделе 3.3. Чем больше API может предоставить данных, которые помогут понять и избежать работы на стороне потребителя, тем лучше. На рис. 5.5 показаны способы добиться этого при проектировании представления REST API для цели «Прочитать счет».

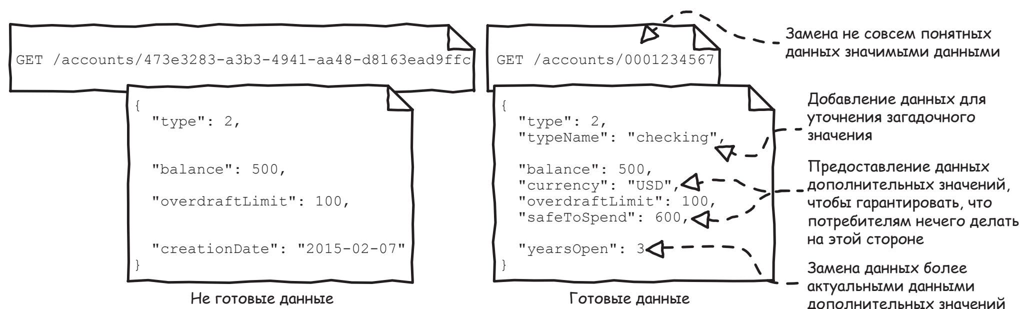



**Рис. 5.5. Упрощаем потребителям работу с готовыми данными**

В предыдущем разделе вы увидели, что было бы лучше создать удобочитаемый тип, такой как `checking`. Но что, если по какой-то причине мы должны использовать числовое именование? В этом случае тип сберегательного счета – это 1, а расчетного – 2. Чтобы уточнить такие числовые значения, можно предоставить дополнительное свойство `typeName`. Его содержание может быть `savings` или `checking`. Таким образом, потребители будут располагать всей необходимой им информацией, чтобы понять, с каким банковским счетом они работают, без необходимости изучать именование или ссылаться на документацию API. Предоставление дополнительной информации определенно помогает уточнить значение загадочных данных.

Функция защиты от овердрафта имеет некоторые ограничения. Если остаток на банковском счете переходит в отрицательное значение после определенного предела (например, 100 долл. США), будут применены штрафы. Это означает, что, если баланс составляет 500 долл., владелец счета может потратить 600. Мы можем предоставить готовое к использованию свойство `safeToSpend`, чтобы потребителю не пришлось выполнять этот расчет, а также можем предоставить информацию о валюте счета, чтобы потребители знали, что суммы для свойств `balance`, `overdraftLimit` и `safeToSpend` указаны в долларах США. Предоставление статических или предварительно рассчитанных данных о добавленной стоимости гарантирует, что потребителям практически ничего не нужно делать или гадать.

Также неплохо будет заменить базовые данные на связанные, но более релевантные данные. Например, свойство `creationDate`, возможно, не представляет особого интереса, поскольку потребитель, скорее всего, захочет узнать лишь, сколько лет открыт счет. В этом случае мы можем предоставить эту информацию напрямую вместо даты создания счета. Таким образом, потребитель получает только актуальные и готовые к использованию данные.

При проектировании REST API каждый ресурс должен быть идентифицирован по уникальному пути (см. раздел 3.2.3). Такие URL-адреса обычно строятся вокруг имен коллекций ресурсов и идентификаторов ресурсов. Чтобы идентифицировать банковский счет, можно было бы использовать URL-адрес `/accounts/{accountId}`. Банковский счет – это элемент, идентифицируемый по `accountId` в коллекции `accounts`. Но что это может быть за accountId? Это может быть технический идентификатор, такой как `473e3283-a3b3-4941-aa48-d8163ead9ffc`, известный как универсальный *уникальный идентификатор* (UUID). Эти идентификаторы генерируются случайным образом, и вероятность генерации одного и того же идентификатора дважды близка к нулю. Таким образом, мы уверены, что путь `/account/473e3283-a3b3-4941-aa48-d8163ead9ffc` уникален, но не можем определить, какой банковский счет обозначает этот URL-адрес, просто прочитав его!

Возможно, мы могли бы взять более удобное для пользователя значение, такое как номер банковского счета, который также является уникальным. Таким образом, URL-адрес для идентификации банковского счета

превращается в /account/{accountNumber}. Путь /account/0001234567 по-прежнему уникален, и теперь у него более понятное значение. Это не зарезервировано для параметра пути – предоставление значимых данных облегчает использование и понимание любого значения.

Как вы убедились, когда вы знаете, о каких аспектах следует позаботиться, проектирование простых представлений относительно просто. Но API – это не только статические данные: люди взаимодействуют с ними для достижения своих целей. И каждое из этих взаимодействий тоже должно быть простым.

## 5.2 Проектирование простых взаимодействий

Чтобы взаимодействовать с объектом или API, пользователи должны предоставить входные данные, чтобы объяснить, что они хотят сделать. В ответ они получают отклик, рассказывающий, как все прошло. В зависимости от того, как проектировщик позаботился о взаимодействии, эти пользователи могут быть абсолютно разочарованы или быть в полном восторге. Стиральные машины являются прекрасным примером для обоих случаев.

Допустим, вы только что вернулись из отпуска, и пришло время стирки. Если вам повезло иметь простую стиральную машину, как показано на рис. 5.6 слева, то задача довольно проста.

**Рис. 5.6.** Простая стиральная машина и сложная

Вы открываете дверцу, кладете белье, добавляете немного порошка и выбираете программу стирки. Для этого вы поворачиваете большую рукоятку – очевидно, с именем **Программа**, – чтобы выбрать тип белья, используя надписи с очевидными названиями, такими как «Шерсть», «Шелк», «Хлопок», «Джинсы», «Рубашки» или «Синтетика». Машина выберет соответствующие циклы очистки/полоскания, температуру воды и скорость отжима в соответствии с типом белья, а также уровень воды в соответствии с весом белья, заданным датчиком веса. При желании вы можете самостоятельно настроить такие параметры, как температура и скорость отжима, используя кнопки или рукоятки с очевидными названиями **Температура** и **Скорость отжима**. Но в этом нет необходимости, поскольку автоматически выбранные параметры являются точными в 95 % всех случаев.

Когда все в порядке, вы нажимаете кнопку запуска. К сожалению, машина не запускается. Она издает несколько звуковых сигналов, и на

ЖК-дисплее отображается сообщение: «Дверца все еще открыта». Вы закрываете дверцу, но машина все равно отказывается запускаться; снова звучит звуковой сигнал, и на дисплее появляется сообщение: «Нет воды». Вы забыли, что отключили воду перед тем, как отправились в отпуск! Открыв главный водяной клапан, вы снова нажмете кнопку запуска. Теперь машина запускается, и на ЖК-дисплее отображается оставшееся время в часах и минутах. Мы можем даже представить, что эта простая стиральная машина сообщает о двух проблемах за один раз, чтобы вы могли решить обе проблемы одновременно.

К сожалению, стирка не всегда так проста. Как показано справа на рис. 5.6, некоторые стиральные машины сложнее в использовании.

Если машина не так проста, вам, возможно, придется предоставить больше информации менее удобным для пользователя способом. Программы, доступные с помощью рукоятки **Prg**, имеют довольно загадочные названия, такие как **Regular**, **Perm. Press** (имеется в виду permanent press) или **Gent. Mot.** (gentler motion). Возможно, датчик веса будет отсутствовать и вам придется нажать кнопку **Half load**, чтобы указать, что машина не заполнена бельем. Не исключено, что вы должны будете самостоятельно выбирать температуру и скорость отжима с помощью ручек **Tmp** и **RPM**. И – удачи в выборе правильной температуры! – от прохладной до очень горячей. После того как вы предоставили все входные данные, обратная связь об ошибках и успехах может оказаться не такой информативной, как у более удобной для пользователя машины. В случае возникновения проблем, таких как «Дверца все еще открыта» или «Нет воды», стиральная машина может просто не запуститься. Если вам повезет, может загореться красная лампочка, но без дальнейших объяснений. И если удастся определить и решить все проблемы, машина может в конечном итоге запуститься, не сообщая вам, сколько времени займет цикл стирки.

Итак, какой тип взаимодействия вы предпочитаете? Простой, требующий минимальных, понятных и простых в предоставлении материалов и показывающий полезную информацию об ошибках и информативный положительный результат? Или сложный, требующий непонятных входных данных и не дающий абсолютного никакого намека на то, что происходит? Вопрос, конечно, чисто риторический. Давайте посмотрим, как можно перенести эту идею прямого взаимодействия на дизайн входных данных и обратной связи при переводе денег для нашего банковского API.

Помните, что в главах 2 и 3 вы узнали, как проектировать цели, параметры и ответы с точки зрения потребителя, избегая точки зрения поставщика? Здесь мы не будем снова обсуждать этот вопрос.

### *5.2.1 Запрос простых входных данных*

Первый этап взаимодействия принадлежит пользователям. Они сами должны предоставить какие-либо входные данные, чтобы сказать, что они хотят сделать. Будучи проектировщиками API, мы можем помочь

им, спроектировав простые вводные данные, подобные тем, что используются в простой в использовании стиральной машине, применяя то, что мы узнали ранее в этой главе.

В нашем банковском API денежный перевод состоит из отправки некой суммы денег с исходного счета на целевой. Перевод может быть мгновенным, отложенным или регулярным. *Мгновенный* перевод, очевидно, выполняется сразу же, в то время как *отложенный* будет выполнен позже. *Регулярный* перевод выполняется несколько раз, с даты начала и до даты окончания, с установленной периодичностью (скажем, еженедельно или ежемесячно). На рис. 5.7 показано, как можно применить то, что мы уже усвоили, для проектирования простых входных данных для перевода денег. Имена должны быть четкими без непонятных сокращений; типы данных и форматы должны быть простыми для понимания, а данные можно легко предоставить.

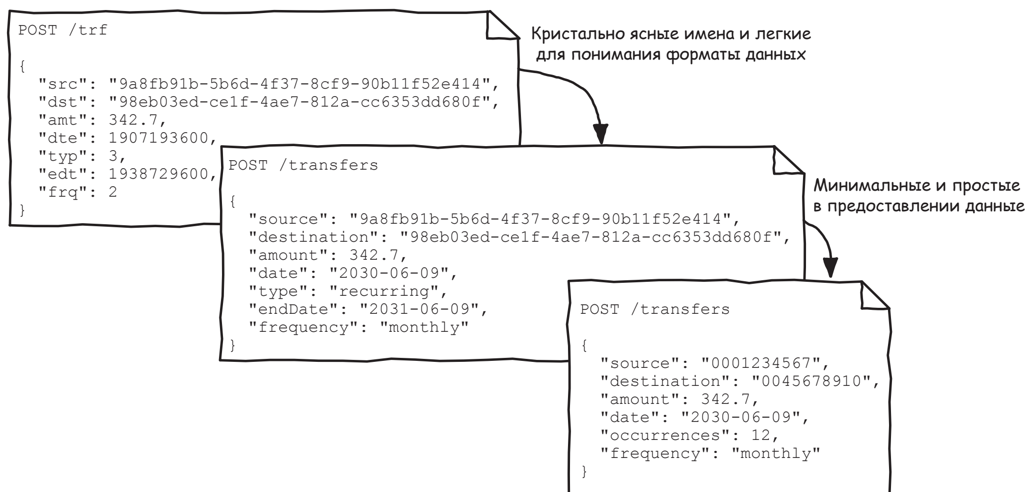



**Рис. 5.7. Проектирование простых входных данных**

Первое из наших правил проектирования простых представлений – использовать кристально ясные имена. Следовательно, цель «Перевод денег» представлена REST-операцией `POST /transfers` (которая создает ресурс перевода) вместо `POST /trf` (сокращение). Мы также используем очевидные имена свойств, такие как `source` и `destination`, вместо `src` или `dst`, например.

Следующее правило – использовать простые для понимания форматы данных. Мы будем избегать использования временных меток UNIX (например, `1528502400`) и станем использовать даты в формате ISO 8601 (например, `2018-06-09`) для свойства `date`, например для обозначения даты отложенного выполнения или даты совершения первого регулярного перевода. Мы также будем избегать использования числовых кодов для таких свойств, как `frequency` и `type`, предпочитая вместо этого удобные для чтения значения, такие как `weekly` или `monthly` и `immediate`, `delayed` или `recurring`.

Можно сделать этот ввод еще более простым, следуя третьему правилу и запрашивая простые в предоставлении данные. Лучше использовать ценные значения, такие как номера счетов для `source` и `destination`, вместо неясных универсальных уникальных идентификаторов. Также, возможно, проще предоставить число для свойства occurrences, вместо того чтобы рассчитывать `endDate` для регулярного перевода. И наконец, можно избавиться от свойства `type`, которое сообщает нам, является ли перевод мгновенным, отложенным или регулярным, потому что серверная часть, получающая запрос, может угадать его значение на основе других свойств.

Таким образом, мы получаем абсолютно простые входные данные. Пользователю все должно быть понятно, когда он будет читать документацию или просматривать пример. И самое главное, эта API-цель очень проста для запуска. Но что происходит, когда пользователь предоставил эти простые входные данные? Давайте рассмотрим вторую часть взаимодействия: обратную связь.

### 5.2.2 Выявление всех возможных ошибок

Первым ответом, который мы получили при использовании стиральной машины, была ошибка. Что это значит для проектирования API? В отличие от того, что мы видели в предыдущих главах, взаимодействие с API не всегда успешно, и мы должны идентифицировать все возможные ошибки для каждой цели. Цель «Перевод денег» также может вызвать такую ошибку, как показано на рис. 5.8.

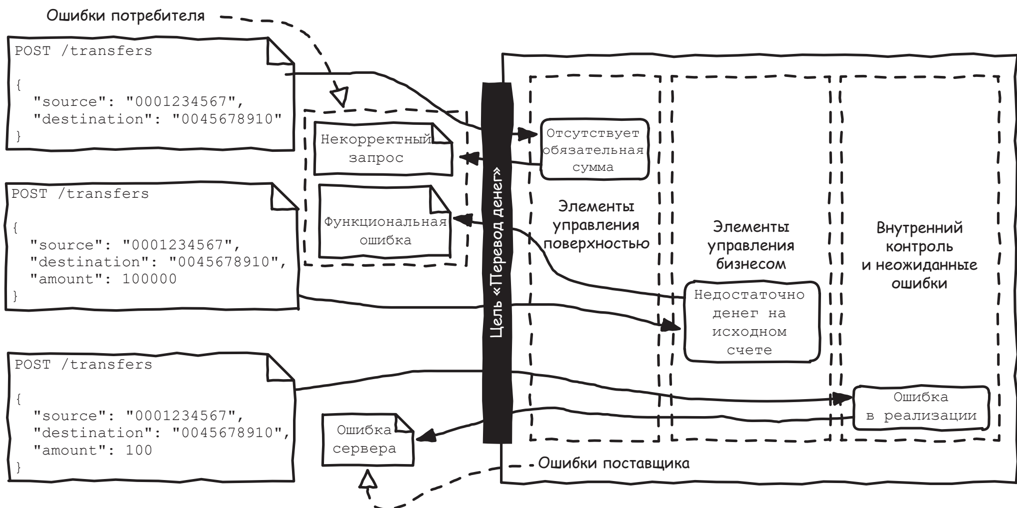



**Рис. 5.8. Некорректный запрос и функциональные ошибки**

Потребители могут получить сообщение об ошибке, если не предоставят обязательное свойство, такое как `amount`, или если предоставят неправильный тип или формат данных, например используя временную метку UNIX вместо строки в формате ISO 8601 для свойства `date`. Такие ошибки известны как *ошибки некорректного запроса*.

Но даже если сервер способен интерпретировать запрос, это не гарантирует, что ответ будет успешным. Сумма денежного перевода может превышать безопасную сумму расходов или максимальную сумму, которую пользователь может перевести в течение одного дня, или может быть запрещено переводить деньги на внешний счет с определенных внутренних счетов.

Такие ошибки являются *функциональными*, которые вызваны бизнес-правилами реализации.

Эти некорректные запросы и функциональные ошибки вызваны потребителями, но поставщик также может инициировать некоторые из них, даже если запрос полностью действителен. Неработающий сервер базы данных или ошибка в реализации могут вызвать ошибку сервера.

Это три разных типа ошибок – некорректный запрос, функциональная ошибка и ошибка сервера. Вы должны определить все возможные ошибки для каждой цели. Обратите внимание, что мы будем рассматривать другие типы ошибок в главах 8 и 10.

Ошибки в результате некорректного запроса могут возникать, когда сервер не может интерпретировать запрос. Поскольку потребители должны отправлять запросы на использование API, такие ошибки могут возникать при любом взаимодействии. Эти ошибки обычно можно идентифицировать сразу после проектирования программного интерфейса. На этом этапе у нас есть подробное представление о запросе, который должен отправить пользователь, и о каждой части запроса, которая может быть причиной такой ошибки.

Функциональные ошибки чаще всего возникают, когда потребители пытаются создать, обновить или удалить данные или инициировать действия. Как правило, их можно идентифицировать после заполнения таблицы целей API, поскольку каждая цель полностью описана с функциональной точки зрения. Для выявления таких потенциальных ошибок не существует волшебного метода; вы должны предвидеть их, пользуясь помощью людей, которые знают бизнес-правила, стоящие за целью. А ошибки сервера могут происходить на каждой цели. С точки зрения потребителя, обычно достаточно определения одной ошибки сервера.

При перечислении ошибок помните, что вы всегда должны ориентироваться на точку зрения потребителя. В каждом из таких случаев, как описано в разделе 2.4, вы должны проверить, относится это к потребителю или нет. Например, при ошибках сервера потребителям просто нужно знать, что их запрос не может быть обработан и что это не их ошибка. Поэтому достаточно одной общей ошибки сервера. Но выявления возможных ошибок недостаточно; нужно спроектировать информативное представление для каждой из них.

### 5.2.3 Возвращение информативного сообщения об ошибке

Проблемы, возникающие с двумя стиральными машинами, решались более или менее легко, в зависимости от того, как была представлена каждая ошибка. Сообщение об ошибке API должно быть максимально ин-

формативным. Оно должно четко сказать потребителям, в чем проблема, и, если возможно, предоставить информацию, которую потребители могут использовать для ее решения, с помощью прямого представления.

В разделе 3.3.1 вы видели, что REST API, полагающийся на протокол HTTP, использует код состояния HTTP, чтобы указать на то, был ли запрос успешным, или нет, как показано на рис. 5.9.

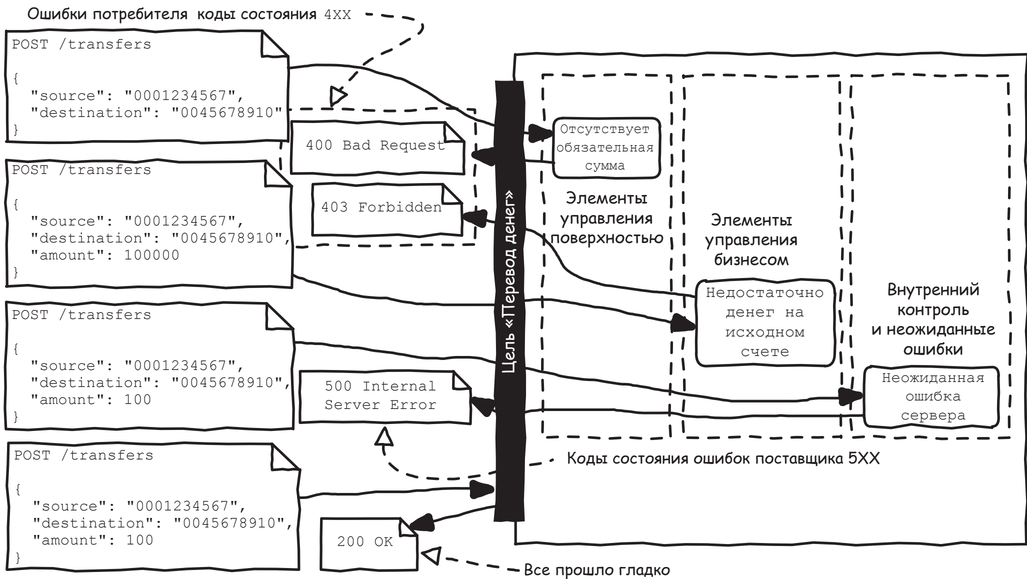



**Рис. 5.9. Выбор точных кодов состояния HTTP**

Вы уже видели, что HTTP-статус `200 OK`, очевидно, означает, что обработка запроса прошла хорошо. Если в запросе отсутствует обязательная сумма, возвращается код состояния `400 Bad Request`. Если сумма слишком большая, это – `403 Forbidden`. А если сервер потерпел катастрофический сбой, возвращается ошибка `500 Internal Server Error`.

Оставим в стороне `200 OK`. Как появились три других кода состояния HTTP? Согласно документу RFC 7231, который описывает «Протокол передачи гипертекста (HTTP / 1.1): семантика и контент» (https://tools.ietf.org/html/rfc7231#section-6), мы должны использовать класс кодов `4XX` для ошибок, вызванных потребителями, и класс `5XX` для ошибок, вызванных поставщиком.

В разделе 3.3.1 вы видели, что REST API, применяющий протокол HTTP, использует код состояния HTTP, чтобы указать, был ли запрос успешным или нет, как показано на рис. 5.9.

> **RFC.** RFC (Request For Comments) – это тип публикации от сообщества разработчиков технологий, используемый для описания интернет-стандартов, но он также может передавать простую информацию и даже экспериментальные новые концепции (которые могут стать стандартом).

Каждый класс кодов содержит базовый код X00; например, 500 Internal Server Error является основным состоянием класса 5XX и идеально подходит для обозначения любого типа ошибки сервера в обобщенном виде. Мы могли бы использовать статус 400 Bad Request для всех выявленных ошибок, вызванных потребителями. Но в этом случае потребители будут знать только, что их запрос недействителен, без каких-либо других намеков на проблему. Было бы интересно иметь возможность различать «Отсутствует обязательная сумма» и «Недостаточно денег на исходном счете». К счастью, протокол HTTP поставляется со множеством кодов 4XX, которые могут быть более точным, чем базовая ошибка 400 Bad Request.

Мы можем сохранить 400 Bad Request, если отсутствует обязательное свойство или есть неверный тип данных. Чтобы уведомить пользователя о том, что мы отказываемся выполнить перевод, превышающий сумму, которую можно безопасно потратить, или о том, что он запросил перевод, который он не имеет права выполнять, можно использовать ошибку 403 Forbidden. Этот код означает, что запрос формально действителен, но не может быть выполнен.

Существует еще один код состояния 4XX, который вы будете часто использовать: это известная ошибка 404 Not Found, ее можно применять для обозначения того, что ресурс не найден. Например, такая ошибка может быть возвращена при запросе GET /accounts/123, если счета 123 не существует.

Существует множество разных кодов состояния HTTP. Вы всегда должны проверять, какой из них наиболее точен при проектировании сообщений об ошибках. В табл. 5.1 показано, как можно использовать некоторые из них.

**Таблица 5.1 Коды состояния HTTP – некорректные запросы и функциональные ошибки**

| Случай использования | Пример | Код состояния HTTP |
|---|---|---|
| Неправильный параметр пути | Чтение несуществующего счета с помощью запроса GET /accounts/123 | 404 Not Found |
| Отсутствует обязательное свойство | Не указана сумма | 400 Bad Request |
| Неверный тип данных | "startDate":1423332060 | 400 Bad Request |
| Функциональная ошибка | Сумма превышает безопасный лимит | 403 Forbidden |
| Функциональная ошибка | Перевод из source в destination запрещен 403 Forbidden | |
| Функциональная ошибка | За последние пять минут уже был выполнен идентичный денежный перевод | 409 Conflict |
| Неожиданная ошибка сервера | Ошибка в реализации | 500 Internal Server Error |

Когда потребители получат один из этих кодов состояния HTTP в качестве сообщения об ошибке, они будут знать, что проблема на их стороне, если код состояния относится к классу 4XX, но у них будет более совершенное представление об источнике проблемы. Если это ошибка 404, они будут знать, что предоставленный URL-адрес не соответствует ни одному из существующих ресурсов и, вероятно, содержит неверный параметр пути. Если они получат ошибку 403, то будут знать, что их запрос формально действителен, но был отклонен из-за неких бизнес-правил. При ошибке 409 они узнают, что их запрос находится в конфликте с предыдущим. А ошибка 400 означает, что их запрос содержит неверные данные или отсутствует обязательное свойство.

Это хорошее начало, но кода состояния HTTP, даже если он правильный, недостаточно. Само по себе состояние HTTP не предоставляет достаточно информации, чтобы помочь решить проблему. Поэтому мы должны также предоставить явное сообщение об ошибке в теле ответа, как показано в приведенном ниже листинге.

**Листинг 5.1. Сообщение об ошибке в теле ответа**

```
{
  "message": "Amount is mandatory"
}
```

Потребитель, который получает код состояния `400 Bad Request` наряду с объектом, содержащим сообщение `Amount is mandatory`, сможет с легкостью решить проблему. Ну что же, человек может легко интерпретировать это сообщение, а как насчет машины?

Допустим, наш банковский API используется в мобильном приложении. Очевидно, что лучше напрямую показать конечному пользователю сообщение `Amount is mandatory`, чем если он просто увидит надпись `Bad Request`. Но не лучше ли выделить поле amount, чтобы помочь конечному пользователю решить проблему? Откуда мобильное приложение может знать, какое значение вызвало проблему? Оно могло бы проанализировать строку сообщения об ошибке, но это было бы довольно неопрятно. Было бы лучше предоставить способ идентифицировать свойство, вызывающее ошибку, программным способом, как показано в приведенном ниже листинге.

**Листинг 5.2. Подробнее сообщение об ошибке**

```
{
  "source": "amount",
  "type": "AMOUNT_OVER_SAFE",
  "message": "Amount exceeds safe to spend"
}
```

Наряду с сообщением об ошибке мы могли бы предоставить свойство `source`, содержащее путь к свойству, вызывающему проблему. В этом случае его значением будет `amount`. Это позволило бы программе определить, какое значение вызывает проблему в случае некорректного за-

проса. Но в случае функциональной ошибки мобильное приложение по-прежнему не будет знать точный тип ошибки. Следовательно, мы также можем добавить свойство `type`, содержащее некий код. В качестве его значения для обозначения суммы, превышающей ту, что пользователь может свободно потратить, может быть, например, `AMOUNT_OVER_SAFE`. Соответствующее сообщение для пользователя может выглядеть так: `Amount exceeds safe to spend`. Подобные действия позволят и людям, и программам, использующим API, точно интерпретировать любые возникающие ошибки.

Как вы видите, мы снова применили принципы прямого представления для проектирования этих ошибок. Обратите внимание, что вам не нужно определять конкретный тип для каждой ошибки; вы можете определить универсальные типы, как показано в приведенном ниже листинге. Например, тип `MISSING_MANDATORY_PROPERTY` может использоваться в любой ошибке для любого отсутствующего обязательного свойства.

**Листинг 5.3. Подробное сообщение об ошибке с использованием универсального типа**

```
{
   "source": "amount",
   "type": "MISSING_MANDATORY_PROPERTY",
   "message": "Amount is mandatory"
}
```

### Более комплексная обработка ошибок

Обратите внимание, что, если ввод более сложный, простого свойства source может быть недостаточно. Например, если мы хотим создать банковский счет с несколькими владельцами, входные параметры могут содержать список owners. Этот список показывает, как можно было бы обработать ошибку в одном из элементов этого списка.

**Листинг 5.4. Подробное сообщение об ошибке с указанием ее источника**

```
{
   "source": "firstname",
"path": "$.owners[0].firstname",
   "type": "MISSING_MANDATORY_PROPERTY",
   "message": "Firstname is mandatory"
}
```

Мы могли бы добавить свойство пути, содержащее путь JSON, чтобы быть более точным. *Путь JSON* (https://goessner.net/articles/JsonPath/) позволяет представить адрес узла в документе JSON. Если у первого владельца в списке отсутствует обязательное свойство `firstname`, значением path будет `$.owners[0].firstname`.

Это лишь несколько примеров информации, которую можно предоставить в качестве ошибок. Вы можете предоставлять столько данных,

сколько необходимо, чтобы помочь потребителям решать проблемы самостоятельно. Например, можно предоставить регулярное выражение, описывающее ожидаемый формат данных в случае ошибки BAD_FORMAT.

Предоставление информативного и эффективного отклика требует, чтобы мы описали проблему и предоставили всю необходимую информацию в удобочитаемом формате как для людей, так и для машин, чтобы помочь потребителям решить проблему самостоятельно (если они могут это сделать). При проектировании REST API это можно сделать с помощью соответствующего кода состояния HTTP и простого тела ответа. Это подходит для сообщения только об одной ошибке за раз. Но что, если проблем несколько?

### *5.2.4 Возвращение исчерпывающего сообщения об ошибке*

В лучшем из возможных сценариев простая стиральная машина сообщает сразу о двух проблемах (открыта дверца и нет воды). Это определенно необходимая функция, если вы хотите создавать удобные в использовании API. Например, запрос о переводе денег может представлять несколько ошибок, связанных с некорректными запросами. Предположим, клиент отправляет запрос, в котором отсутствуют значения свойств source и destination. Сначала он получит сообщение об ошибке, уведомляющее его о том, что обязательное свойство source отсутствует. После исправления этой ошибки он сделают еще один вызов и получит еще одну ошибку, сообщающую о том, что отсутствует свойство destination. Это отличный способ расстроить потребителей. Всю эту информацию можно было предоставить в первоначальном сообщении об ошибке!

Чтобы избежать слишком большого числа циклов «запрос–ошибка» и гнева со стороны потребителей, лучше всего возвращать как можно более исчерпывающие сообщения об ошибках, как показано в приведенном ниже листинге.

**Листинг 5.5. Возвращение нескольких ошибок**

```
{
  "message": "Invalid request",
  "errors": [
    {
      "source": "source",
      "type": "MISSING_MANDATORY_PROPERTY",
      "message": "Source is mandatory" },
    {
      "source": "destination",
      "type": "MISSING_MANDATORY_PROPERTY"},
      "message": "Destination is mandatory"
    }
  ]
}
```

Поэтому мы должны сразу вернуть список `errors`, содержащий ошибки, которые появились в результате двух некорректных запросов. Каждая ошибка может быть описана, как вы видели ранее в листинге 5.3. То же самое относится и к случаям, когда запрос не является некорректным, но содержит несколько функциональных ошибок.

Что произойдет, если есть оба типа ошибок? На рис. 5.10 показано, что может произойти в этом случае.

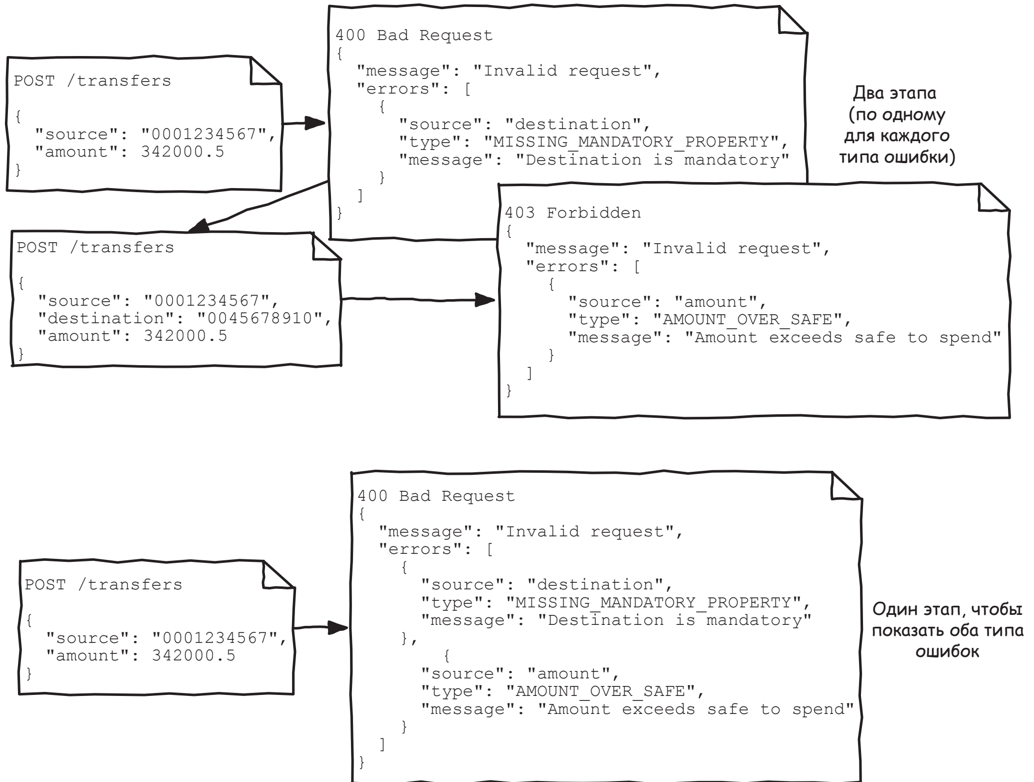



**Рис. 5.10.** Обработка различных типов ошибок

В этом примере запрос содержит `source` и сумму (`amount`), превышающую ту, что пользователь может свободно потратить, но отсутствует `destination`.

При проектировании отдельных ошибок мы решили использовать `400 Bad Request` для некорректного запроса данного типа и `403 Forbidden` для функциональных ошибок. Можно было бы разделить два эти типа ошибок и вернуть сообщение об ошибке, чтобы сначала сообщить об отсутствии `destination`, а затем вернуть второе сообщение о функциональной ошибке (`Amount exceeds safe to spend`). Но имеет ли это смысл для потребителей? Им и вправду есть дело до этого различия? Вероятно, нет, по крайней мере, в конкретном случае использования, где мы предоставляем всю необходимую информацию для определения типа ошибки.

Я бы порекомендовал вернуть общий `400 Bad Request`, содержащий все некорректные запросы и функциональные ошибки. Однако обратите внимание, что это решение не может быть универсальным средством. Вам придется проанализировать вашу конкретную ситуацию, чтобы сделать наиболее подходящий выбор, когда придет время решать – разделять категории ошибок или нет.

Группировка нескольких ошибок в одном сообщении упрощает взаимодействие, уменьшая количество циклов «запрос–ошибка». Но если вы проектируете REST API, это означает использование общего состояния HTTP и использование данных ответов для предоставления подробной информации о каждой ошибке. Как только все проблемы будут решены, взаимодействие должно закончиться сообщением об успешном результате.

### 5.2.5 Возвращение информативного сообщения об успешном результате

В случае использования стиральной машины мы увидели, что предоставление информативного сообщения об успешном результате может быть очень полезным для пользователей. И в самом деле очень полезно знать, когда закончится стирка. Аналогичным образом, подобного рода сообщения в случае с API должны предоставлять полезную информацию потребителям помимо простого подтверждения. Как добиться этого? Применить то, что о чем мы уже узнали из этой главы!

При использовании стиля REST API информативное сообщение об успехе может опираться на те же вещи, что и сообщения об ошибках: точный код состояния HTTP и простое тело ответа, как показано на рис. 5.11.

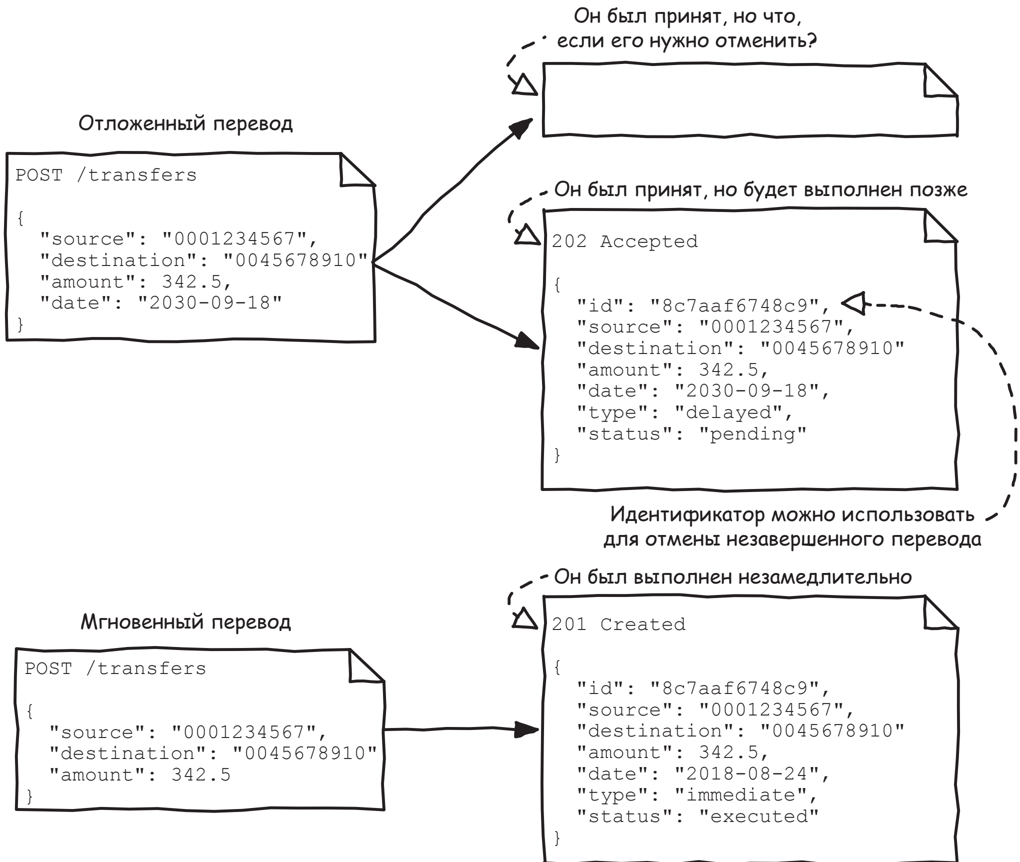



**Рис. 5.11.** Полностью информативное сообщение об успехе

Как указано в RFC 7231 (https://tools.ietf.org/html/rfc7231#section-6.3), «…класс кода состояния 2xx (успешно) указывает на то, что запрос клиента был успешно получен, понят и принят». Таким образом, в случае любого успешного результата мы могли бы вернуть 200 OK. Но, как и в классе 4XX, для класса 2XX существует множество кодов, которые могут более точно описать, что произошло в некоторых случаях использования.

Если денежный перевод является мгновенным, можно было бы вернуть код состояния 201 Created. Это означает, что перевод был создан. В случае отложенного перевода можно было бы вернуть ответ 202 Accepted, указывающий на то, что запрос на перевод денег был принят, но еще не выполнен. В этом случае подразумевается, что он будет выполнен в запрошенный день. То же самое касается *регулярного* перевода: мы можем использовать ответ 202 Accepted, чтобы сообщить потребителям, что переводы будут выполнены, когда это должно будет произойти. Но опять же, состояния HTTP недостаточно; простое и информативное сообщение гораздо полезнее.

Такой ответ должен содержать каждый фрагмент информации созданного ресурса, как вы уже узнали в предыдущих главах. Возвращать свойства, рассчитанные сервером (например, тип перевода или его статус), интересно, так же как и время, когда должна закончиться стирка. ID особенно интересен потребителям, которым может потребоваться отменить конкретный отложенный перевод, только что ими созданный. Без него они просто не смогут этого сделать.

Таким образом, в основном информативные сообщения об успешном результате предоставляют информацию о том, что произошло, а также дают информацию, способную помочь на следующих этапах. Давайте резюмируем правила, которые мы определили для проектирования простых взаимодействий:

- входные и выходные данные должны быть простыми;
- должны быть выявлены все возможные ошибки;
- сообщение об ошибке должно объяснить, в чем проблема, и помочь потребителям решить ее самостоятельно;
- следует избегать сообщения о нескольких ошибках по одной;
- сообщение об успешном результате должны предоставить информацию о том, что было сделано, и дать информацию, чтобы помочь на следующих этапах.

Теперь мы можем проектировать простые индивидуальные взаимодействия. Но будут ли эти взаимодействия формировать простой поток при совместном использовании?

## *5.3 Проектирование простых потоков*

Чтобы использовать объект или API, пользователю может потребоваться объединить несколько взаимодействий. Удобство использования сильно зависит от простоты этого потока взаимодействий.

Если вы находитесь на 5-м этаже здания и хотите отправиться на 16-й, то можете использовать одну из четырех лифтовых кабин. На рис. 5.12 показаны различные возможные варианты вашего путешествия на лифте.

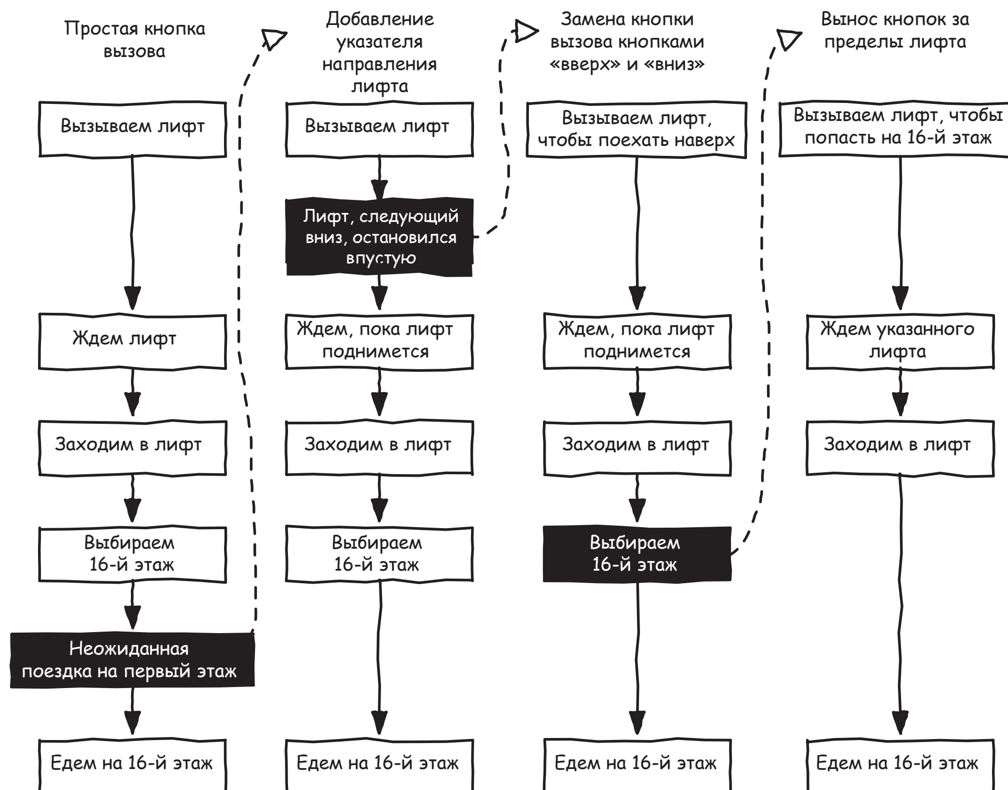



**Рис. 5.12.** Улучшение потоков использования лифта

Если это базовая система, на стене есть одна кнопка вызова для всех кабин. Она загорается, когда вы нажимаете на нее. Потом вы ждете, не зная, какая из кабин лифта приедет. Когда одна из них прибывает, раздается сигнал. Вы входите и нажимаете кнопку с обозначением 16-го этажа. К сожалению, этот лифт шел вниз на первый этаж. Поэтому вы спускаетесь вниз и после этого поднимаетесь на 16-й.

Когда вы заходите в кабину лифта, не зная о том, куда она поедет, это может раздражать. К счастью, производители лифтов улучшили свои системы, чтобы избежать такой ситуации, добавив световой сигнал или ЖК-экран снаружи каждой кабины лифта, чтобы показать, идет ли она вверх или вниз. Это лучше, но зачем останавливать лифт, который спускается вниз, тем, кто хочет поехать наверх? Это неудобно для тех, которые ждет прибытия лифта, а также для тех, кто находится внутри кабины. Эту болевую точку можно устранить, заменив одну кнопку вызова двумя: **вверх** и **вниз**. Теперь вы можете вызвать лифт, чтобы поехать наверх или вниз, и на вашем этаже остановятся только кабины, идущие в этом направлении.

Но когда вы входите в кабину лифта, вам все равно нужно нажать вторую кнопку, чтобы уточнить, на какой этаж вы хотите попасть. В некоторых системах кнопки **вверх** и **вниз** заменены кнопками с обозначением

этажей, которые вы видите в кабине лифта. Теперь, собираясь вызвать лифт, чтобы попасть на 16-й этаж, вы просто нажимаете кнопку с обозначением этого этажа, и на ЖК-экране будет показано, какую кабину лифта использовать.

Как видите, поток взаимодействия, для того чтобы попасть на 16-й этаж, был упрощен за счет улучшения обратной связи и входных данных, предотвращения ошибок и даже агрегирования действий. Этот поток взаимодействия стал абсолютно простым. Давайте посмотрим, как можно применить эти принципы для создания простого потока взаимодействия при передаче денег с помощью банковского API.

### 5.3.1 Построение простой цепочки целей

Мы увидели, как улучшение входных данных и обратной связи путем добавления указателя направления и замены кнопки вызова кнопками **вверх** и **вниз** помогло улучшить последовательность действий, необходимых, для того чтобы попасть на 16-й этаж здания. Заботясь подобным образом о входных данных и ответных сообщений в нашем API, мы можем построить простую цепочку целей.

Цепочка существует, только если ее звенья связаны. Когда потребители используют API для конкретной цели, у них должны быть все данные, необходимые для ее выполнения. Такие данные могут быть известны самим потребителям или могут быть предоставлены результатами предыдущих целей, о чем вы узнали в разделе 2.3. Частичная таблица целей API, показанная на рис. 5.13, предоставляет такую информацию.

| Кто | Что | Как | Входные данные (источник) | Выходные данные (использование) | Цели |
|---|---|---|---|---|---|
| Потребители | Перевести деньги со счета на собственный или внешний счет | Перечислить счета | | Список счетов (денежный перевод) | Перечислить счета |
| | | Список предварительно зарегистрированных бенефициаров | | Список предварительно зарегистрированных бенефициаров (денежный перевод) | Перечислить бенефициаров |
| | | Перевести деньги | Исходный счет (Перечислить счета), счет назначения (Перечислить счета, Перечислить бенефициаров), сумма (потребитель) | Отчет о переводе | Перевести деньги |

Рис. 5.13. Таблица с целями банковского API

Эта таблица говорит нам, что для мгновенного перевода денег потребители должны предоставить сведения о сумме (`amount`), исходном счете (`source`) и счете, на который осуществляется перевод (`destination`). Очевидно, что потребители знают, сколько денег они хотят перевести. Исходный счет должен быть одним из счетов, который они могут получить с помощью цели «Перечислить счета». Если API используется приложением мобильного банкинга, эти счета принадлежат лицу, использующему приложение. Конечным счетом должен быть один из этих счетов

(например, для перевода денег с текущего счета на сберегательный) или предварительно зарегистрированный внешний бенефициар (например, для отправки денег другу или оплаты аренды квартиры).

Если вам интересно, почему банковская компания заставляет своих клиентов предварительно регистрировать бенефициаров, это делается как в целях безопасности, так и в целях удобства использования. Для регистрации внешнего бенефициара необходима двухфакторная аутентификация с использованием обычного пароля и SMS-подтверждения, электронной почты или токена безопасности, генерирующего случайные пароли, – таким образом, имеется гарантия, что только фактический клиент может сделать это. И как только бенефициар будет зарегистрирован, сделать денежный перевод довольно просто: не нужно запоминать и внимательно набирать номер счета, на который осуществляется перевод, и нет необходимости в двухфакторной аутентификации. Потребители могут использовать цели «Перечислить счета» и «Перечислить бенефициаров», чтобы получить информацию о возможных исходных и конечных счетах, прежде чем они переведут деньги. Итак, у нас есть цепочка.

Но цепочка сильна настолько же, насколько ее самое слабое звено. Каждое взаимодействие, участвующее в потоке, должно быть простым. Это то, что вы узнали ранее в разделе 5.2. На рис. 5.14 показан поток денежных переводов.

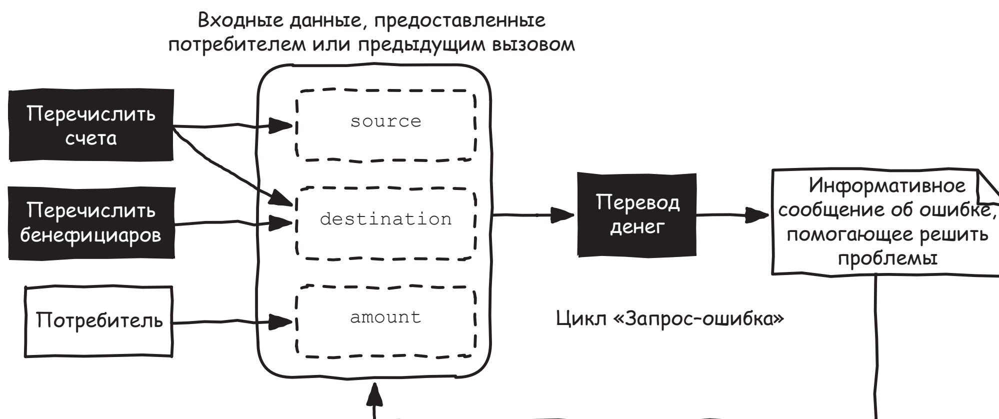



Рис. 5.14. Денежный перевод

Цели «Перечислить счета» и «Перечислить бенефициаров» довольно просты, потому что они не требуют входных данных и не возвращают ошибок. Входные данные для цели «Перевести деньги» просты, но эта цель может возвращать много разных ошибок. Если бы мы возвращали только сообщение об ошибке `400 Bad Request`, потребителям было бы трудно выполнить перевод денег. Но благодаря тому, что вы почерпнули в этой главе, теперь вы знаете, что должны предоставлять информативные и исчерпывающие сообщения об ошибках, чтобы помочь потребителям решать проблемы, с которыми они сталкиваются. Это значитель-

но уменьшит количество циклов запросов и ошибок и позволит избежать искусственного расширения длины цепочки вызовов API.

Таким образом, первый шаг на пути к простой цепочке целей API состоит в том, чтобы запросить простые входные данные, которые могут быть предоставлены потребителями или другой целью в цепочке, и возвращать исчерпывающие и информативные сообщения об ошибках, чтобы ограничить циклы запросов и ошибок. Теперь мы должны быть в состоянии построить прямую цепочку целей. Но разве нельзя сделать это короче и более плавно, предотвращая ошибки?

### 5.3.2 Предотвращение ошибок

В примере с лифтом, когда мы добавили указатель поворота, это помогло предотвратить неожиданное путешествие на первый этаж. Предотвращение ошибок – хороший способ сгладить и сократить поток целей API. Но как предотвратить ошибки? Применяя один из принципов простого представления – предоставляя готовые к использованию данные. Мы должны проанализировать каждую ошибку, чтобы определить, можно ли ее предотвратить, предоставив некие данные до этой цели.

Цель денежного перевода может привести к появлению различных функциональных ошибок:

- сумма превышает безопасный лимит;
- сумма превышает совокупный дневной лимит перевода;
- исходный счет нельзя использовать в качестве источника перевода;
- данный вариант назначения нельзя использовать с этим источником.

Попробуем предотвратить ошибку «Исходный счет нельзя использовать в качестве источника перевода». Мы могли бы добавить логическое свойство `forbiddenTransfer` к каждому счету, полученное с помощью цели «Перечислить счета». Таким образом, потребитель сможет предоставить исходный счет, когда для этого свойства установлено значение false, при запросе денежного перевода. Но это значит, что ему придется выполнить фильтрацию на своей стороне. На рис. 5.15 показана более подходящая альтернатива.

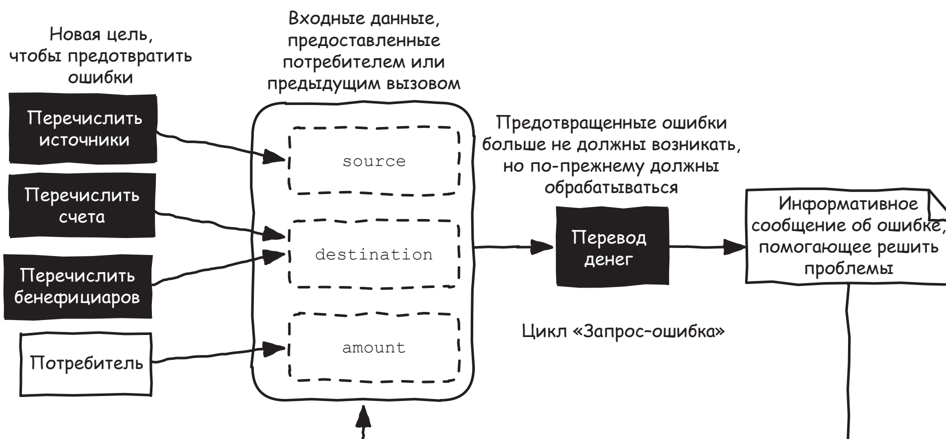



**Рис. 5.15.** Предотвращение ошибок в потоке денежных переводов

Здесь добавлена новая цель «Перечислить источники», возвращающая только те счета, которые можно использовать в качестве источника для перевода денег. Эта цель также может возвращать максимально допустимую сумму для перевода, основанную на безопасных расходах и совокупных суточных лимитах перевода, чтобы предотвратить соответствующие ошибки. Потребитель может использовать эти готовые значения для реализации элементов управления поверхностью на своей стороне.

Довольно неплохо. С помощью этой новой цели можно предотвратить три из четырех ошибок! Но учтите, что любая предотвращенная ошибка все равно должна обрабатываться целью «Перевод денег». Некоторые потребители не могут реализовать элементы управления поверхностью или вызвать эту цель напрямую без предоставления надлежащих параметров.

Как видите, предотвращение ошибок может сделать поток целей более плавным. Помните, что это можно сделать с помощью:

- анализа возможных ошибок для определения дополнительных данных, которые могут их предотвратить;
- улучшения сообщений существующих целей об успехе для предоставления таких данных;
- создания новых целей для предоставления таких данных.

## Ограничения REST: код по требованию

Обратите внимание, что, если ввод более сложный, простого свойства source может быть недостаточно. Например, если мы хотим создать банковский счет с несколькими владельцами, входные параметры могут содержать список owners. Этот список показывает, как можно было бы обработать ошибку в одном из элементов этого списка.

На сайте ограничение «код по требованию» выполняется, когда веб-сервер предоставляет файлы JavaScript, содержащие код, который выполняется в браузере. Можно попытаться сделать это и с помощью API, но это будет означать, что вы предоставляете код, понятный для всех потребителей, независимо от языка программирования, с помощью которого они созданы. Такой сценарий кажется довольно нереальным, но «своего рода кода по требованию» можно добиться, предоставляя адекватные данные с помощью специальных вспомогательных целей или регулярных целей, что мы только что делали для перевода денег. Многие бизнес-правила могут быть представлены более или менее сложными данными, которые потребители смогут использовать в качестве кода.

И в самом деле, поток целей был улучшен. Но эффективно ли и ориентировано ли это на потребителя, когда мы вызываем цели «Перечислить счета» и «Перечислить бенефициаров», чтобы узнать все возможные значения `destination`?

### *5.3.3 Объединение целей*

Размещение кнопок с обозначением этажей за пределами кабины лифта позволило заменить вызов лифта и выбор 16-го этажа одним действием: вызовом лифта, чтобы отправиться на 16-й этаж. Такие объединения могут быть полезны для оптимизации потока целей API.

Возможно, вас беспокоило, что значение `destination` может быть получено либо из цели «Перечислить счета», либо из цели «Перечислить бенефициаров». Такой дизайн можно рассматривать как свидетельство точки зрения поставщика, поскольку он требует от потребителей работы на своей стороне. И если принять во внимание, что некоторые ассоциации «источник–назначение» запрещены, становится ясно, что это прекрасный пример точки зрения поставщика. Как показано на рис. 5.16, эту проблему можно решить, создав цель «Перечислить варианты назначения для источника», которая заменяет цели «Перечислить счета» и «Перечислить бенефициаров», в качестве источника для свойства `destination`.

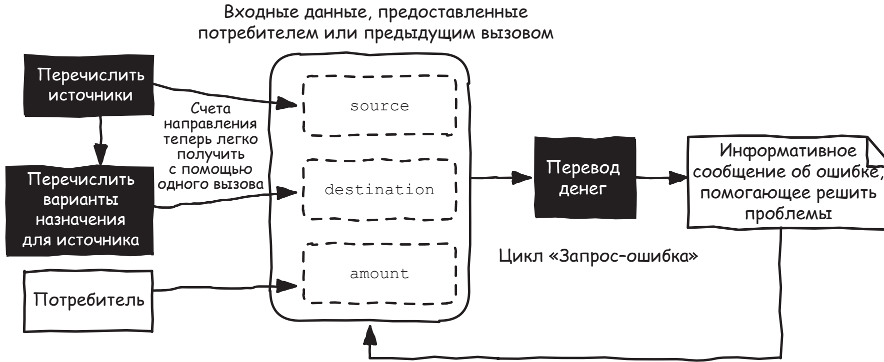



**Рис. 5.16.** Для составления списка вариантов назначения из выбранного источника требуется один вызов

Новая *агрегированная* цель возвращает только возможные варианты назначения для данного источника, причем источник извлекается с помощью цели «Перечислить источники». Эта новая цель упростит поток целей, и тут есть бонус! Это также предотвращает появление ошибки «Данный вариант назначения нельзя использовать с этим источником». Теперь у потребителей меньше целей для использования, и у них есть доступ ко всему, что нужно, чтобы избежать сообщений об ошибках от цели «Перевод денег».

Это все, что мы можем сделать? На рис. 5.17 показана одна последняя оптимизация.

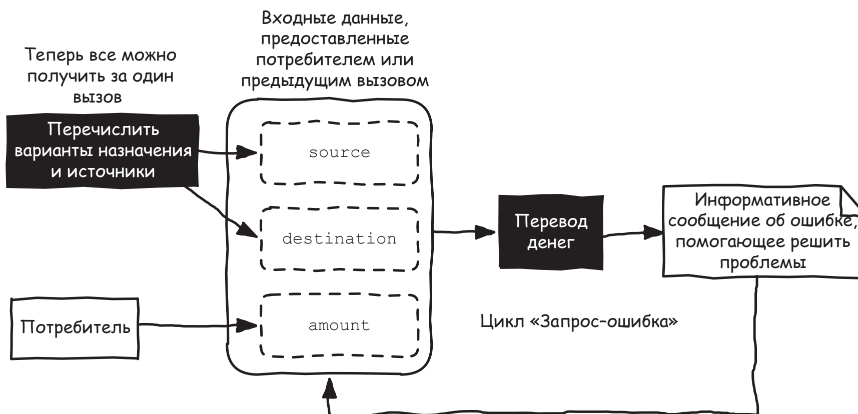



**Рис. 5.17** Один вызов предоставляет все данные, необходимые для выбора источника и варианта назначения

Поскольку количество возможных комбинаций «источник вариант назначения» относительно ограничено, мы можем предоставить все возможные ассоциации «источник вариант назначения» с помощью одной цели «Перечислить варианты назначения и источники», которая объединяет цели «Перечислить источники» и «Перечислить варианты назначения для источника». Это не обязательно, но возможно.

Имейте в виду, что такие объединения должны выполняться только в том случае, если конечные цели действительно имеют смысл для потребителя с функциональной точки зрения. Также имейте в виду, что такие объединения могут привести к проблемам с производительностью; мы поговорим на эту тему в главах 10 и 11. На данный момент есть еще одна вещь, которую стоит упомянуть, чтобы создать полностью прямые потоки.

### 5.3.4 Проектирование потоков без сохранения состояния

Эта тема отсутствует в примере с лифтом из реальной жизни; она берет свое начало из ограничений REST, которые вы видели в разделе 3.5.2.

Представим себе следующий рабочий процесс для инициирования денежного перевода.

1. Перечислить источники.
2. Перечислить варианты назначения для выбранного источника (источник хранится в сеансе на стороне сервера).
3. Передать деньги по месту назначения (используемый источник хранится в сеансе на сервере).

Такой поток фиксирует состояние, и это определенно не очень хорошая идея; его ни за что нельзя проектировать или реализовывать. И в самом деле, цель перевода средств нельзя использовать отдельно, поскольку она основана на данных, хранящихся в сеансе благодаря предыдущим вызовам. Некоторые потребители вполне могут выбрать источник и вариант назначения самостоятельно, не используя цель «Перечислить варианты назначения». Каждая цель должна быть пригодна для использования без других целей, а все необходимые входные данные должны быть явно объявлены.

#### Ограничения REST: отсутствие состояния

Отсутствие состояния достигается путем сохранения контекста на сервере между запросами (используя сеанс) и только полагаясь на информацию, предоставленную вместе с запросом для его обработки.

Это гарантирует, что любой запрос может быть обработан любым экземпляром реализации API вместо конкретного экземпляра, содержащего данные сеанса. И это также способствует независимому использованию целей API и, следовательно, облегчает их повторное использование в различных контекстах.

Итак, для того чтобы спроектировать совершенно простые потоки, вы должны следовать этим правилам:

- убедитесь, что каждая цель обеспечивает прямое взаимодействие;
- убедитесь, что входные и выходные данные согласованы между вызовами цели;
- по возможности предотвращайте ошибки, добавляя данные к существующим целям для создания новых целей;

- по возможности объединяйте цели, но только если это имеет смысл для потребителя с функциональной точки зрения;
- каждая цель цепочки должна быть без фиксации состояния.

На этом все, что касается проектирования простого API! В следующей главе мы продолжим изучение юзабилити, чтобы научиться проектировать предсказуемые API, которые можно использовать инстинктивно.

## *Резюме*

- Любое представление должно быть легко понятно для людей и программ.
- Любое представление должно быть максимально информативным.
- Сообщение об ошибке должно содержать достаточно элементов, чтобы можно было понять и, возможно, решить проблему.
- Сообщения об успехе должны описывать, что было сделано.
- Потоки целей можно оптимизировать путем добавления данных или целей для предотвращения ошибок.
- Потоки целей могут быть упрощены путем объединения целей, но только если это имеет смысл с функциональной точки зрения.

# 6 Проектирование предсказуемого API

**В этой главе мы рассмотрим:**

- согласованность при создании интуитивно понятных API;
- добавление функций для упрощения использования и адаптации к пользователям;
- добавление метаданных и метацелей, чтобы направлять пользователей.

В предыдущей главе мы начали наше путешествие, чтобы узнать, как создавать удобные в использовании API, и познакомились с фундаментальными принципами, которые можно использовать для создания API, простых для понимания и простых в использовании. Это неплохо – теперь мы знаем, как спроектировать достойный API. Но можно сделать еще лучше. Как насчет проектирования *потрясающего* API? Как насчет проектирования API, который люди смогут использовать инстинктивно, не задумываясь об этом, даже если они используют его впервые? Как это сделать?

Вы когда-нибудь испытывали огромное удовольствие, когда использовали незнакомый предмет или приложение впервые? Ну знаете, когда все настолько интуитивно и легко, что чувствуешь себя невероятно умным, открывая все его возможности самостоятельно? Это возможно не только потому, что вы на самом деле невероятно умны, но и потому, что вещь, которую вы используете, была спроектирована, чтобы сделать ее полностью предсказуемой. Конечно, не все способны дать такое потря-

сающее чувство, но каждый день вы можете сталкиваться с ситуациями, когда предсказуемость помогает вам, даже если вы этого не понимаете.

Почему вы знаете, как открыть дверь в здании, в котором вы никогда не были? Потому что она похожа на те двери, которые вы видели раньше. Как использовать банкомат в стране, где говорят на языке, который вы не понимаете? Потому что он адаптирует свой интерфейс под вас. Как не заблудиться в огромной и запутанной станции метро? Там есть знаки, указывающие, куда идти.

Можно ли и в самом деле спроектировать такие интуитивно понятные API? Да, можно! Как и любой объект, API может быть предсказуемым, потому что он имеет схожие черты, с которыми сталкивались другие пользователи ранее, потому что он может адаптироваться к желанию пользователей или потому что он предоставляет информацию, чтобы направлять их.

## 6.1 Согласованность

Если вам встретится стиральная машина, подобная той, что показана на рис. 6.1, с кнопкой, на которой изображен треугольный значок, повернутый вправо, вы можете легко угадать назначение этой кнопки. Почему? Потому что вы уже видели этот значок на различных медиапроигрывателях.

**Рис. 6.1.** Стиральная машина и кассетный плеер с одинаковым значком

С середины 1960-х годов такое обозначение использовали все медиаплееры. Начиная от плееров DCC (Digital Compact Cassette) и заканчивая проигрывателями компакт-дисков и программными мультимедийными проигрывателями, каждое из этих устройств использует один и тот же значок в виде треугольника для кнопки **Начать воспроизведение**. Поэтому можно догадаться, что эта кнопка запускает стиральную машину.

> **ПРИМЕЧАНИЕ.** *Согласованный* дизайн лишен изменений или противоречий, что помогает сделать интерфейс интуитивно понятным, используя преимущества предыдущего опыта взаимодействия с пользователями.

Я уверен, что вы также знаете, как выглядит стандартная кнопка **Пауза**. Что, если медиаплеер не использует стандартный значок для этой

кнопки? Пользователи будут озадачены, и им придется постараться понять, как приостановить воспроизведение аудио или видео.

**ПРИМЕЧАНИЕ.** *Несогласованный* дизайн вносит изменения или противоречия, которые затрудняют понимание и использование интерфейса.

Опять же, то, что справедливо в отношении реальных пользовательских интерфейсов, справедливо и для API. Важно поддерживать согласованность дизайна API, чтобы он был предсказуемым. Это можно сделать, если применить немного дисциплины, обеспечивая согласованность данных и целей внутри и во всех API, используя и соблюдая предписанные стандарты, а также бесстыдно копируя других. Но если согласованность может привести к потрясающему дизайну, это нельзя использовать в ущерб *юзабилити*.

### 6.1.1 Проектирование согласованных данных

Данные являются основой API – ресурсы, параметры, ответы и их свойства формируют API. И все их значения, имена, типы, форматы и организация должны быть согласованы, чтобы потребители могли легко понять их. Итак, проектирование согласованных API начинается с выбора согласованных имен, как показано на рис. 6.2.

Согласованное именование
Несогласованное именование
Согласованное именование
Несогласованное именование

| | | | | | |
|---|---|---|---|---|---|
| Получить счета | accountNumber | accountNumber | Получить счета | balanceDate | balanceDate |
| Получить счет | number | accountNumber | Получить счет | dateOfCreation | creationDate |
| Перевести деньги | source | sourceAccountNumber | Перевести деньги | executionDay | executionDate |

**Рис. 6.2.** Несогласованное и согласованное именование

В плохо спроектированном банковском API номер счета можно представить в виде свойства `accountNumber` для результата цели «Получить счета», в виде свойства `number` для получения счета и свойства `source` для вводных данных цели «Перевод денег». Здесь та же часть информации в трех разных контекстах представлена совершенно другими именами. Пользователи не смогут с легкостью установить связь между ними.

Как только пользователи замечают номер счета в виде `accountNumber`, они ожидают увидеть часть информации, всегда обозначаемую `accountNumber`. Люди привыкли к единообразию в дизайне. Поэтому, является ли это свойство частью подробной распечатки счетов, сводкой списка счетов или оно используется в качестве параметра пути, номер счета должен называться `accountNumber`.

При выборе имен для различных представлений одного и того же понятия позаботьтесь, чтобы они были похожи. Когда речь идет о переводе денег для идентификации исходного счета, оригинальное имя должно оставаться узнаваемым, но его можно изменить, чтобы предоставить

больше информации о природе свойства; можно было бы назвать его `sourceAccountNumber`, например. Теперь потребители могут установить связь между этими свойствами и предположить, что они представляют одно и то же понятие.

Это также подходит и для несвязанных данных аналогичного типа или для представления схожих концепций. Например, `balanceDate`, `dateOfCreation` и `executeDay` обозначают дату, но первое свойство использует суффикс `Date`; второе – префикс `dateOf`, а третье – суффикс `Day`. Использование *общего* суффикса или префикса в имени для предоставления дополнительной информации о характере того, чему мы даем это имя, – неплохая практика, пока это делается согласованно. Здесь (на рис. 6.2), используется один и тот же суффикс `Date` для всех дат, но можно выбрать другое решение, если вы действуете согласованно. Однако даже при правильном названии свойство, например, по-прежнему может подвергаться несогласованности, как показано на рис. 6.3.

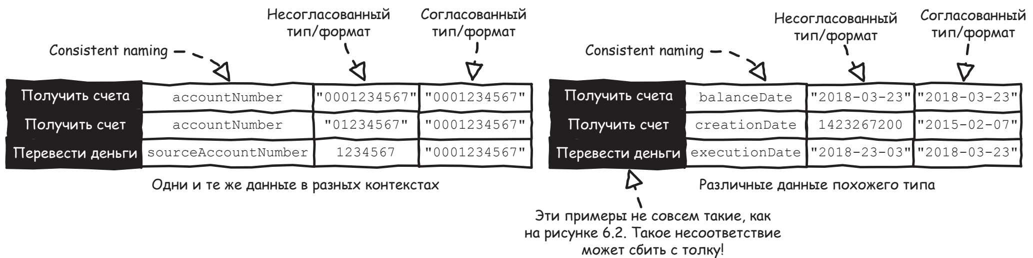



**Рис. 6.3. Несогласованные и согласованные типы данных и форматы**

Номер счета может быть представлен в виде строки «`0001234567`» в результате цели «Получить счета», строки «`01234567`» для цели «Получить счет» и в виде числа `1234567` для входных данных цели «Перевод денег». Такие изменения неизбежно вызовут ошибки на стороне потребителя. Чтобы исправить их, потребители должны стандартизировать эти представления и знать, когда преобразовать один тип или формат в другой для использования его в данном контексте.

Люди и программное обеспечение не хотят, чтобы такие несоответствия вызывали у них удивление. Как только потребители увидят первое свойство `accountNumber` в виде строки с определенным форматом, они ожидают, что все другие представления номеров счетов будут строками с таким же форматом. Даже если у них разные имена, разные представления одного и того же понятия должны использовать один и тот же тип и формат.

Выбор типа данных или формата также может оказать общее влияние и на API. Как, по вашему мнению, потребители будут реагировать, если они увидят свойство `balanceDate` банковского счета в виде строки в формате ISO 8601 (например, `2018-03-23`), `creationDate` счета в виде временной метки UNIX (например, `1423267200`), а `ExecutionDate` перевода в виде даты в формате ГГГГ-ДД-ММ (например, `2018-23-03`)? Им это не понравится, потому что тут нет согласованности.

Люди стремятся к глобальному единообразию в дизайне. Когда потребители видят одно из свойств даты и времени в виде строки в формате ISO 8601, они ожидают, что все свойства даты и времени будут строками в формате ISO 8601. Как только вы выбрали формат для типа данных, он должен использоваться для всех представлений одного и того же типа данных.

Потребители стремятся к глобальному единообразию во всех аспектах API, а не только в том, что касается типов данных и форматов. В чем заключается проблема с URL-адресами `/accounts/{accountNumber}`, обозначающими счет, или `/transfer/{transferId}`, которые обозначают денежный перевод? Это `/accounts` против `/transfer` – множественное число против единственного. Как только потребители освоятся с использованием имен во множественном числе для коллекций, они ожидают, что все коллекции будут с именами во множественном числе. Вы можете использовать единственное число, если хотите, но, независимо от вашего выбора, придерживайтесь его! И это касается не только URL-адресов, но и каждого выбранного вами имени и значения.

**ПРИМЕЧАНИЕ.** Соглашения об именах могут быть определены для имен свойств, имен параметров запроса, кодов, моделей JSON Schema в файле OpenAPI и т. д. Как только вы выберете соглашение об именовании, строго следуйте ему.

Итак, в чем же проблема с двумя URL-адресами и структурами данных, показанными на рис. 6.4? Их организации данных несогласованны.

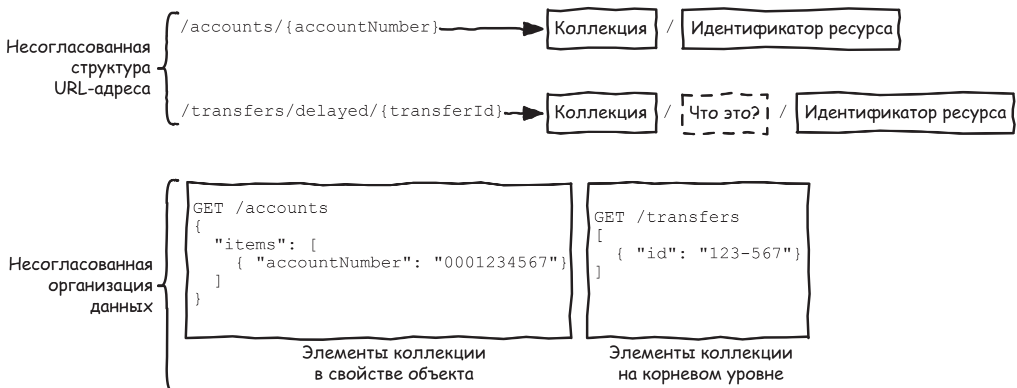



**Рис. 6.4.** Несогласованная организация

У URL-адресов `/account/{accountNumber}` и `/transfer/delayed/{transferId}` разная организация. `/transfer/delayed/{transferId}` вводит неожиданный уровень между именем коллекции и идентификатором ресурса, делая URL-адрес труднее для понимания. Вместо этого можно было бы использовать `/delayed-transfer/{transferId}`, например.

Каждый уровень URL-адреса всегда должен иметь одинаковое значение. Когда потребители привыкают к шаблону организации данных, они

ожидают, что он будет использоваться повсеместно. Опять же, это относится не только к URL; организация данных во входных и выходных данных также может представлять шаблоны. В нижней части рис. 6.4 элементы двух коллекций представлены двумя разными способами. Если каждый спроектированный вами ресурс коллекции представлен объектом, содержащим свойство `items`, являющееся массивом, не вздумайте проектировать его как простой массив. Почему? Потому что потребители будут удивлены этим изменением.

После выбора соглашений об организации данных строго соблюдайте их. По сути, каждый бит данных API должен быть согласованным. Но API-интерфейсы состоят не только из статических данных; они созданы, для того чтобы что-то делать, и все поведение API также должно быть согласованным.

### 6.1.2 Проектирование согласованных целей

Поведение API определяется его целями: они ожидают входных данных и возвращают сообщения об успехе или ошибках, и все эти цели можно использовать для формирования различных потоков. Очевидно, что все это должно быть согласованным.

В чем состоит проблема с целями «Прочитать счет» и «Получить информацию о пользователе»? Эти два названия несогласованны – они обозначают один и тот же тип действия, но используют разные глаголы. Разумнее было бы назвать их «Прочитать счет» и «Прочитать информацию о пользователе», особенно если они должны быть представлены в виде функций в коде, таких как `readAccount()` и `readUserInformation()`.

Будем надеяться, что в случае с REST API программное представление этих целей будет волшебным образом согласовано благодаря использованию протокола HTTP. Обе эти цели будут представлены запросом `GET /resource-path` с использованием одного и того же метода HTTP (на разных путях).

Как вы видели в разделе 6.1.1, данные должны быть согласованными, как и входные данные цели. Например, при перечислении транзакций на счете с помощью запроса `GET /accounts/{accountId}/transactions` может быть полезно иметь возможность получать только те транзакции, которые произошли между двумя датами. Такие параметры запроса, как `fromDate=1423267200` и `tillDay=2015-03-17`, могут делать свое дело, но они явно несогласованны. Было бы лучше использовать `fromDate=2015-02-07` и `toDate=2015-03-17`. При проектировании входных данных цели вы должны использовать согласованные имена, типы данных, форматы и организацию.

То же самое касается сообщений, возвращаемых в случае успеха или ошибки. Если все цели, ведущие к созданию чего-либо, возвращают код состояния `200 OK` без использования таких кодов, как `201 Created` или `202 Accepted`, было бы разумно избегать введения новых целей, возвращающих разные успешные коды состояния HTTP вместо обычного ответа `200 OK`. Вы можете использовать только небольшое подмножество всех существующих кодов состояния HTTP; это может иметь смысл в не-

которых контекстах. Но даже если потребители должны рассматривать любое неожиданное состояние класса `2XX` как `200 OK`, такая несогласованность некоторых из них может удивить.

Согласованность также имеет значение, когда речь идет о сообщениях об ошибках. Очевидно, что вы должны возвращать согласованные коды состояния HTTP для обозначения ошибок, но возвращаемые информативные данные также должны быть согласованными. Вы научились проектировать такие данные в разделе 5.2, и, если вы определили универсальные коды, такие как `MISSING_MANDATORY_PROPERTY`, чтобы указать, что обязательное свойство отсутствует, всегда используйте этот код в своем API.

Наконец, согласованность касается не только внешнего вида API, но и его поведения. Если все ранее спроектированные важные действия (например, денежный перевод) состоят из <действия> по управлению и цели «Выполнить <действие>» – в первом случае выполняются все возможные проверки без выполнения действия, а во втором фактически выполняется действие, – все новые важные действия должны быть обозначены целями, имеющими такое же поведение. При проектировании API вы также должны позаботиться о создании согласованных потоков целей.

Таким образом, каждый аспект контракта интерфейса, каждое поведение API должно быть согласованным. Но мы говорили только о согласованности в API; на самом деле это только первый уровень последовательности в проектировании API.

### 6.1.3 Четыре уровня согласованности

Глядя на кнопки пульта дистанционного управления телевизора, вы видите, что они согласованы; например, кнопки с цифрами имеют одинаковую форму. Если вы посмотрите на пульт телевизора и Blu-Ray- или DVD-плеер одного и того же производителя, у них могут быть небольшие различия, но в основном они согласованны, поскольку обычно имеют общие характеристики (те же самые кнопки с цифрами, например). Если вы посмотрите на какое-нибудь мультимедийное устройство или мультимедийный проигрыватель, они тоже будут согласованными, потому что у них имеются одинаковые те же элементы управления, в особенности кнопка воспроизведения. Тут снова могут быть небольшие различия, но вы по-прежнему чувствуете себя комфортно, переходя с одного устройства на другое. И наконец, если вы столкнулись с кнопкой запуска на стиральной машине, вы знаете, каково ее назначение, потому что, вероятно, видели ее раньше – возможно, на разных медиаплеерах. Эти примеры демонстрируют четыре уровня согласованности, которые можно применять при проектировании API:

- *уровень 1* – согласованность в рамках API;
- *уровень 2* – согласованность API-интерфейсов организации/компании/команды;
- *уровень 3* – согласованность с областью (областями) API;
- *уровень 4* – согласованность с остальным миром.

Мы только что видели первый уровень, где API должен быть согласован с самим собой, предлагая согласованные данные, цели и поведение. Каждый раз, когда вы выбираете дизайн, вы должны убедиться, что он не внесет изменения в API или, что еще хуже, противоречие. При рассмотрении API в целом потребители должны видеть обычный интерфейс. Переходя с одной части на другую, они должны чувствовать, что эта новая часть знакома. Они должны быть в состоянии определить, как она работает, даже если они никогда раньше не использовали ее.

Так же как согласованность важна в рамках одного API, она важна для всех API, которые предоставляет организация. Это второй уровень согласованности. В организации может быть одна команда с одним проектировщиком API или несколько команд со множеством проектировщиков. Потребителей API организации не волнует, были ли эти API созданы одним проектировщиком или несколькими. Их волнует то, что у API есть общие функции, поэтому они могут с легкостью понимать и использовать любую часть API, после того как научились работать с ней.

Как разные цели в рамках одного API должны иметь общие функции, так и разные API в организации также должны иметь общие функции. Совместное использование общих функций (таких как организация данных, типы данных или форматы) улучшает взаимодействие между API. Взять данные из API и передать их другому будет проще, если функции совместимы.

Третий уровень касается согласованности с областью (областями) API или с теми, которые используются им. Например, если вы просто хотите получить адреса каких-то клиентов или выдать отформатированные адреса для печати на конвертах, представление адреса будет осуществляться по-разному. Если нужно рассчитать расстояния в API для морской навигации, вы будете использовать морские мили, а не мили или километры. Обычно существуют стандартные или, по крайней мере, общепринятые практики, которые необходимо соблюдать при работе с конкретной областью.

И наконец, четвертый и последний уровень: API должны быть согласованны с остальным миром. Существуют общепринятые практики – стандарты, если хотите, – которые можно использовать. Если вы будете следовать им, это не только сделает ваши API предсказуемыми для людей, которые никогда ранее не использовали их, тем самым повышая их совместимость с остальным миром, но также облегчит работу проектировщику. Давайте посмотрим, как это можно сделать.

### 6.1.4 *Копируя других: следование общепринятым практикам и соблюдение стандартов*

Зачем изобретать велосипед, если кто-то это уже сделал? Существуют тысячи стандартов, которые вы можете использовать в своем API, тысячи API-интерфейсов, чьи проектировщики используют общепринятые методы, и несколько справочных API-интерфейсов, которые можно копировать без зазрения совести. Вы знаете значение символов воспроиз-

ведения и паузы, показанных на рис. 6.5, потому что вы встречали их на разных устройствах.

**Рис. 6.5.** Символы воспроизведения и паузы, определенные стандартом ISO 7000

Возможно, вы узнали их значение, прочитав руководство пользователя первого устройства, с которым вы столкнулись, но после этого каждый раз, когда вы видели эти символы, вы могли догадаться, что они значат. На каждом устройстве, где они используются, их назначение одинаково.

Внешний вид и значение символов воспроизведения и паузы определяются стандартом ISO 7000 (https://www.iso.org/obp/ui/#iso:pub:PUB400008:en). После того как пользователи увидели их на одном из устройств, они могут определить их назначение на любом другом устройстве. Они могут использовать новое устройство без предварительного опыта работы с ним, потому что у них есть опыт работы с другими устройствами, использующими те же стандарты. Любой проектировщик, желающий создать кнопки **Пуск** и **Пауза**, вероятно, будет использовать эти символы, вместо того чтобы изобретать новые. Как и любое реальное устройство, API может использовать преимущества стандартов (в широком смысле), чтобы его было легче понять.

Наш банковский API может предоставлять информацию о суммах в разных валютах. Создание нашей собственной классификации валют и ее постоянное использование во всех наших банковских API-интерфейсах – это хорошо. Таким образом, мы, по крайней мере, действуем согласованно в рамках нашей организации. Но было бы лучше использовать международный стандарт ISO 4217 (https://www.iso.org/iso-4217-currency-codes.html), который позволяет обозначать валюты с помощью трехбуквенного (USD, EUR) или трехзначного кода (840, 978). Используя такой стандарт, мы можем не отличаться от остального мира! Любой, кто когда-либо использовал стандарт ISO 4217 где-то еще, поймет значение валютных кодов ISO 4217 без необходимости изучать нестандартную классификацию. Точно так же стандарт ISO 8601, который мы видели ранее для обозначения значений даты и времени, является не только удобным для человека форматом, но также широко применяется в индустрии программного обеспечения.

**ПРИМЕЧАНИЕ.** Использование стандартов облегчает понимание, потому что потребители, возможно, уже знакомым с этими значениями. Это также повышает функциональную совместимость вашего API, потому что его данные будут легко использоваться другими API, использующими те же стандарты.

Существуют стандарты для форматов данных, именования и организации данных и даже процессов. И не все из них определены ISO; существует множество других организаций, определяющих стандарты и рекомендации, которые можно использовать при проектировании API. Используйте вашу любимую поисковую систему и найдите что-то вроде «стандарт <данных>» или «формат <данных>», и вы, вероятно, найдете формат, который можно использовать для представления этих «<данных>». Попробуйте, например, выяснить, как обозначать телефонные номера.[1]

Но быть *стандартом* не всегда означает следовать спецификациям ISO или другой организации. Если наш банковский API представляет отложенный перевод с помощью URL-адреса `/delayed-transfer/{transferId}`, можно догадаться, что использование HTTP-метода `DELETE` приведет к отмене перевода. Если вы получите ответ `410 Gone`, можно догадаться, что отложенная передача была выполнена или отменена до того, как вы попытались ее удалить. Откуда вам это известно? Потому что вы ожидаете, что банковский API, который утверждает, что является REST API, будет строго следовать протоколу HTTP, определенному в RFC 7231 (https://tools.ietf.org/html/rfc7231).

HTTP-метод `DELETE` может использоваться в ресурсе для удаления или отмены концепции, представленной URL-адресом. Ответ `410 Gone` довольно явный; согласно стандарту, он «…указывает на то, что запрошенный ресурс больше не доступен и уже не будет доступен». И далее, он «…должен использоваться, если ресурс был удален намеренно, и должен быть очищен».

Таким образом, REST API могут быть согласованными, просто применяя правила протокола HTTP к письму. Таким образом, каждый может быстро начать использовать любой REST API.

## Ограничения REST: единообразие интерфейса

Архитектурный стиль REST гласит, что «…все взаимодействия должны руководствоваться концепцией идентифицированных ресурсов, которыми манипулируют посредством представления состояний ресурсов и стандартных методов». *Стандартный метод* – в действительности мощная концепция, которая помогает обеспечить согласованность. В основном весь протокол HTTP (особенно HTTP-методы, а также коды состояния HTTP) обеспечивает согласованную структуру для REST API, что делает их полностью предсказуемыми.

В мире проектирования API существуют общепринятые практики, которым можно следовать, например как та, что показана на рис. 6.6.

[1] Следует искать формат E.164, рекомендованный ITU-T (Сектор стандартизации электросвязи Международного союза электросвязи).

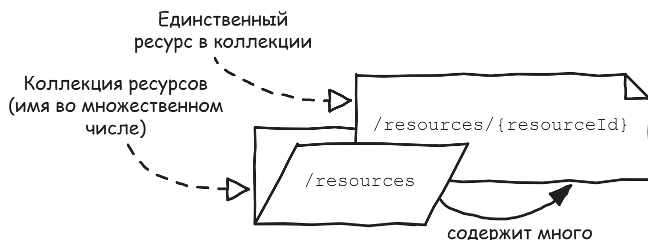



**Рис. 6.6. Общепринятый шаблон URL-адреса**

Как вы видели в разделе 3.2.3, хотя для структуры URL-адреса REST API и не существует стандартных правил, многие из них используют шаблон `/resources/{resourceId}`. Здесь resources – это коллекция, обозначенная существительным во множественном числе. Она содержит элементы типа `resource`.

Даже если в мире проектирования API не все стандартизировано, существуют общепринятые практики, которые очень близки к стандартам. Желательно следовать им, чтобы гарантировать, что ваш API будет понятен потребителями на основе их опыта работы с другими популярными API.

И наконец, во многих случаях можно просто скопировать то, что сделали другие. Зачем беспокоиться о переосмыслении параметров нумерации с нуля, когда эта проблема уже решалась многими другими разработчиками API? Можно посмотреть на некоторые хорошо известные API и повторно использовать тот дизайн, который вы предпочитаете. Это упростит вам жизнь как проектировщику API, и, если ваши пользователи станут применять эти API, им будет комфортно, когда они будут работать с ними впервые. Все в выигрыше.

Вот практический сценарий. Допустим, вам нужно спроектировать API, который обрабатывает изображения, чтобы создать подходящую палитру цветов для веб-разработчиков. Пользователи, предпочитающие бесплатный вариант, могут запрашивать до десяти палитр в месяц; если они хотят больше, то должны будут купить подписку. И те и другие могут отправлять не более одного запроса в секунду. Размер отправленных изображений не может превышать 10 Мб. Рассмотрим следующие вопросы:

- Как бы вы обозначали цвета стандартным способом, который соответствует потребностям веб-разработчиков?
- Какие явные коды состояния HTTP можно было бы использовать, чтобы сообщить
  - пользователям, предпочитающим бесплатный вариант, что они должны заплатить, чтобы получить больше?
  - что у пользователей превышена квота запросов в секунду?
  - пользователям, что изображение слишком большое?
- Какой RFC (запрос комментариев) можно было бы использовать для создания простого сообщения об ошибках?

- Бонус: полностью спроектируйте и опишите этот API, используя спецификацию OpenAPI.

Как показал этот раздел, быть согласованным в рамках одного API и в других API – это хорошо. Это уменьшает необходимость упражняться с ними, чтобы использовать их эффективно; все, что нам нужно делать, – это следовать общепринятым стандартам или практикам и даже (без зазрения совести или нет) копировать другие. Это также делает наши API совместимыми, особенно при использовании традиционных стандартов. И наконец, приятный бонус – это облегчает нашу работу как разработчиков API, поэтому нам не нужно терять время, изобретая велосипед. Кажется, что согласованность упрощает все; но, к сожалению, это не всегда так.

### *6.1.5 Согласованность – это сложно, и все нужно делать по-умному*

Вы должны знать о двух вещах, касающихся согласованности: быть согласованным трудно и согласованность нельзя применять вслепую. Что касается согласованности между API, просто необходимо следовать одним и тем же соглашениям при проектировании разных API. Это также требует знаний о том, какие API существуют на самом деле, что на удивление сложно и требует дисциплины.

Вы должны формально определить свой дизайн с помощью правил в документе, который называется «Руководство по проектированию API» или «Руководство по стилю проектирования API». Даже если вы – единственный проектировщик API в своей компании, такие рекомендации важны, потому что со временем мы склонны забывать, что делали ранее (даже в одном API). Определение таких рекомендаций облегчает не только стандартизацию всей поверхности API, но и труд проектировщика API. В главе 13 вы узнаете, как создавать такие рекомендации.

Вам также потребуется доступ к существующим проектированию, чтобы сохранить согласованность. Как только у вас появится шпаргалка по разработке API и каталог API, вы сможете сосредоточиться на решении реальных проблем и не тратить время на то, чтобы заново изобретать колесо, которое создали несколько месяцев назад.

Согласованность – вещь хорошая, но не в ущерб удобству использования или здравому смыслу. Иногда вы будете понимать, что, если зайдете слишком далеко, это разрушит гибкость и сделает процесс проектирования чрезвычайно сложным, что приведет к созданию согласованных, но совершенно непригодных API. Важно понимать, что иногда вы можете быть несогласованными из-за заданного контекста в угоду юзабилити (как мы уже видели в разделе 3.4, когда говорили о компромиссных решениях). В главе 11 вы также обнаружите, что единого способа создания API не существует: проектирование API требует от нас адаптации к контексту.

Поэтому быть согласованным – отличный способ стать предсказуемым, и это помогает потребителям интуитивно использовать ваш API. Но вы также можете схитрить и позволить людям выбирать то, что они хотят получить, используя ваш API, что делает его еще более предсказуемым.

## 6.2 Адаптируемость

Покупая книгу через интернет-магазин, вы часто можете приобрести ее в разных версиях. Она может быть продана в виде печатной или электронной книги или аудиокниги. Она также может продаваться на разных языках, как, например, франкоязычный вариант этой книги, *Le Design des APIs Web*, который я надеюсь однажды подготовить. Все эти версии являются различными представлениями одной и той же книги; вы должны указать, какая версия нужна вам, когда добавляете книгу в корзину.

А когда вы получаете заказанную книгу, то обычно не читаете все сразу. Вы читаете ее постранично, а останавливаясь, отмечаете последнюю прочитанную страницу. Продолжая читать, вы сразу переходите к этой странице, не просматривая книгу с самого начала. Когда вы читаете некоторые виды книг, особенно технические, то можете перейти непосредственно к определенной главе или разделу, а следовательно, к определенной странице, поэтому можете не читать главы в естественном порядке. Вы также можете брать только части, относящиеся к определенной теме.

Управление различными представлениями одной и той же концепции и обеспечение частичного, выбранного или адаптированного представления некоего контента не зарезервировано для книг; это же можно делать и с API. Мы можем создать такой *адаптируемый* дизайн API, который поможет сделать API предсказуемым, а также удовлетворит разных пользователей. Если потребители могут указать, что им нужно, они могут предсказать, что они получат.

Здесь мы обсуждаем три распространенных способа создания адаптивного дизайна API: предоставление и принятие различных форматов; интернационализация и локализация; и обеспечение фильтрации, нумерации страниц и сортировки. Это не полный список опций – вы можете найти другие и даже создать свой собственный список, когда это необходимо.

### 6.2.1 Предоставление и принятие разных форматов

JSON – очевидный способ обозначения списка транзакций на счете в нашем банковском API. Как показано в левой части рис. 6.7, список транзакций может быть массивом объектов JSON, каждый из которых состоит из трех свойств: `date`, `label` и `amount`.

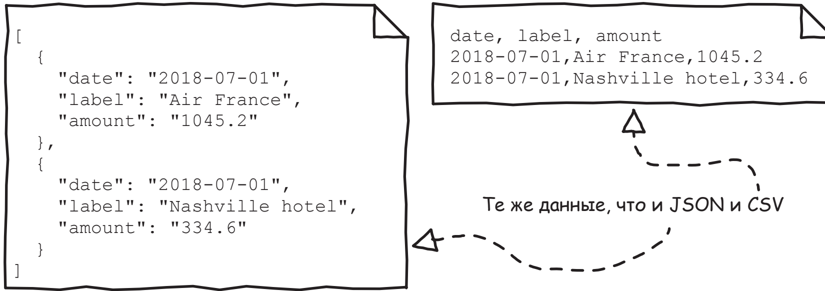



**Рис. 6.7. Список транзакций в формате JSON и CSV**

Но JSON – не единственный вариант. В правой части рисунка показаны те же данные, представленные в формате CSV. В этом случае список транзакций представлен строками текста. Каждая строка, обозначающая транзакцию, состоит из трех значений, разделенных запятыми (,): первое – это дата, второе – метка, а третье – сумма. Этот список транзакций может быть представлен и в виде PDF-файла. Вы также можете позволить потребителям выбирать формат в зависимости от их потребностей. Единственное ограничение – ваше воображение.

## Ограничения REST: единообразие интерфейса

Архитектурный стиль REST гласит, что все взаимодействия должны руководствоваться концепцией идентифицированных ресурсов, которыми манипулируют через *представления* состояний ресурсов и стандартные методы и которые предоставляют все метаданные, необходимые для понимания представлений и знания того, что можно делать с этими ресурсамиа.[1] Один ресурс может быть предоставлен потребителем и возвращен поставщиком во многих разных форматах с помощью, например, различных представлений, таких как JSON или CSV. Это обеспечивает мощный механизм, который позволяет REST API адаптироваться к своим потребителям и, следовательно, быть предсказуемым. Возвращенные заголовки также предоставляют информацию о фактическом формате возвращаемого представления.

[1] *Рой Томас Филдинг*, «Архитектурные стили и проектирование сетевых программных архитектур», 2000 г. (https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2).

Но если цель «Получить счет» может возвращать список в различных форматах, как потребителям определить, какой формат нужен? На рис. 6.8 показаны два разных способа сделать это.

Как мы видим слева на рис. 6.8, можно было бы добавить параметр `format` к этой цели, чтобы позволить потребителям уточнить, нужен ли им список транзакций в формате JSON, CSV или PDF. Например, чтобы получить список в виде CSV-документа, потребители могут отправить запрос `GET`, который определяет `format=CSV`:

```
GET /accounts/{accountId}/transactions?format=CSV
```

Это возможно, но поскольку банковский API – это REST API, мы также могли бы воспользоваться преимуществами протокола HTTP и использовать *согласование содержимого*.

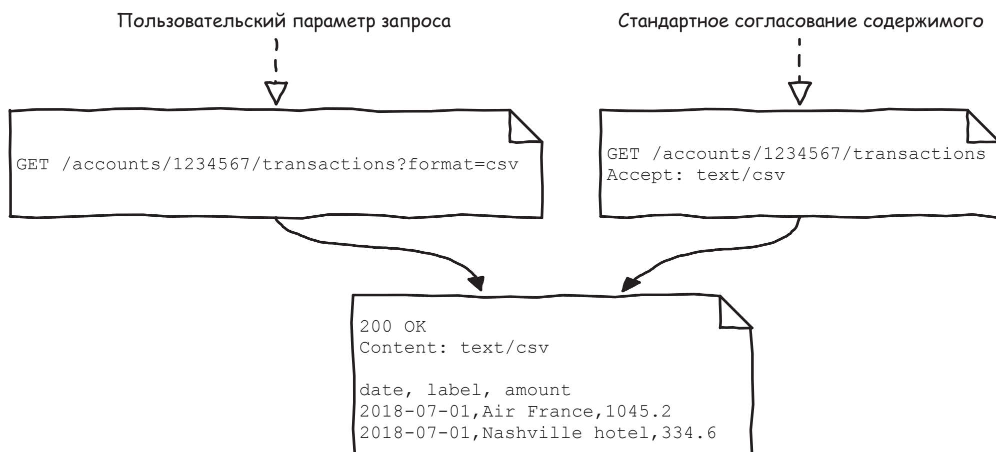



**Рис. 6.8.** Два варианта запроса списка транзакций в виде CSV-документа

При отправке запроса `GET /accounts/{accountId}/transactions` на сервер API потребители могут добавить HTTP-заголовок `Accept: text/csv` после HTTP-метода и URL-адреса, чтобы указать, что они хотят, чтобы этот список транзакций был представлен в виде данных CSV. (Этот подход показан справа на рис. 6.8.) Если все в порядке, сервер API выдает в ответ код состояния `200 OK`, за которым следует заголовок `Content-type: text/csv` и список транзакций в виде CSV-документа.

Потребители также могут отправить `Accept: application/json` или `Accept: application/pdf` для получения данных в формате JSON или PDF-файла соответственно, при этом сервер возвращает ответ с заголовком `Content-type: application/json` или `Content-type: application/pdf` с последующим документом в соответствующем формате. В этом примере представлены две новые функции протокола HTTP: HTTP-заголовки и согласование содержимого. Давайте рассмотрим их повнимательнее.

*Заголовки HTTP* – это разделенные двоеточиями пары имя–значение. Их можно использовать как в запросах, так и в ответах для предоставления дополнительной информации. В запросе эти заголовки располагаются после строки запроса, содержащей метод HTTP и URL-адрес. В ответе они располагаются после строки состояния, содержащей код состояния HTTP и поясняющую фразу. Существует около 200 различных стандартных заголовков HTTP, и даже можно создать свой собственный, если это необходимо. Они используются для различных целей, одной из которых является согласование содержимого.

*Согласование содержимого* – это механизм HTTP, позволяющий обмениваться различными представлениями одного ресурса. Когда HTTP-сервер (а следовательно, и сервер REST API) отвечает на запрос, он должен

указывать тип носителя возвращаемого документа. Это делается в заголовке ответа `Content-type`. Большинство REST API использует медиатип `application/json`, поскольку возвращаемые документы являются документами JSON. Но потребители могут предоставить заголовок запроса `Accept`, содержащий медиатип, который они хотят получить. Как показано на рис. 6.9, в банковском API три возможных медиатипа для списка транзакций на счете – это `application/json`, `application/pdf` и `text/csv`.

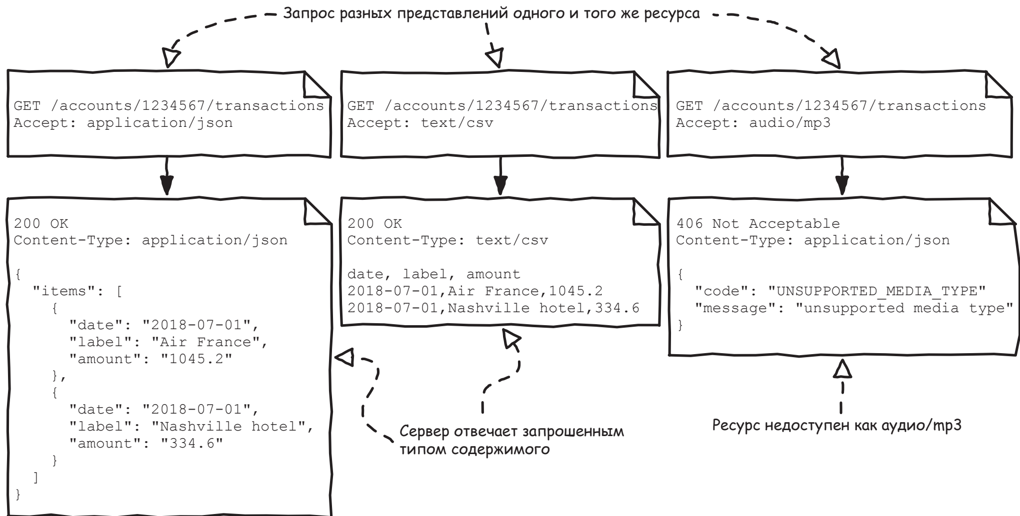



**Рис. 6.9. Запрос трех разных представлений списка транзакций**

Если потребитель запрашивает такой медиатип, как `audio/mp3`, который не обрабатывается поставщиком, сервер ответит ошибкой `406 Not Acceptable`. Обратите внимание, что запрос без заголовка `Accept` подразумевает, что потребитель будет принимать любой медиатип. В данном случае сервер вернет представление по умолчанию – например, данные в формате JSON.

Это также подходит, когда потребитель должен предоставить данные в теле запроса. В разделе 5.2.1 в предыдущей главе вы увидели, что для создания перевода потребители должны отправить запрос `POST /transfers`, тело которого содержит исходный счет, конечный счет и сумму. Предполагалось, что это тело будет документом JSON, но это также может быть и другим медиатипом. Например, потребитель может отправить XML-документ, содержащий информацию, необходимую для создания перевода,[1] для этого он должен предоставить заголовок `Content-type: application/xml`. Если сервер API не способен понять XML, он возвращает ошибку `415 Unsupported Media Type`. Если потре-

[1] XML (расширяемый язык разметки) – это язык разметки, который должен быть понятен как человеку, так и машине. Это был стандарт де-факто для API и веб-сервисов до появления JSON. В XML такое свойство, как сумма, будет представлено в виде `<amount>123.4</amount>`.

бители также хотят получить результат в виде XML-документа вместо документа в формате JSON, они должны предоставить заголовок `Accept: application/xml` вместе с заголовком `Content-type`, тем самым сообщая серверу: «Я отправляю вам XML и хотел бы, чтобы при ответе вы также использовали XML».

Это замечательно – согласование содержимого, независимо от того, обеспечивается ли он используемым протоколом или обрабатывается вручную, позволяет потребителям выбирать формат, который они хотят использовать при взаимодействии с API, при условии, что он поддерживается. Но это еще не все.

### 6.2.2 Интернационализация и локализация

Даже переведенная на французский язык, электронная книга *Le Design des APIs* – это еще одно представление одно и той же книги. Как применить эту концепцию к примеру с банковским API?

В разделе 5.2.3 вы научились проектировать простое сообщение об ошибках. Например, когда клиент пытается совершить денежный перевод, API может вернуть ошибку с сообщением `Amount exceeds safe to spend`. Такое сообщение могут увидеть все конечные пользователи – но что, если они не понимают по-английски? Разработчики, создающие приложение или сайт с использованием банковского API, должны будут управлять переводом этого сообщения на своей стороне. С технической точки зрения это возможно, поскольку эта ошибка идентифицируется с помощью четко идентифицируемого типа `AMOUNT_OVER_SAFE`. Но, возможно, мы, проектировщики API, можем помочь разработчикам, использующим наш API, и предложить способ получать сообщения об ошибках на других языках, помимо английского.

Мы могли бы добавить параметр language ко всем целям банковского API, значение которого должно быть языковым кодом ISO 639 (http://www.loc.gov/standards/iso639-2/php/code_list.php). Например, код `fr` означает французский язык, а `en` – английский. В разделе 6.1.4 вы узнали, что использование стандартов полезно, чтобы гарантировать, что значение будет легко понятным и совместимым. Но подождите – `en` просто означает *английский язык*. Британский английский и американский английский можно считать разными языками, так же как французский язык во Франции и французский канадский. Поэтому ISO 639 – не очень хорошая идея; было бы лучше использовать более точный стандарт для определения языка.

RFC 5646 (https://tools.ietf.org/html/rfc5646), который определяет языковые метки, – тот стандарт, который мы ищем. В этом формате используются языковые коды ISO 639 и коды стран ISO 3166: английский язык США – `en-US`, британский английский – `en-UK`, французский язык Франции – `fr-FR`, французский канадский – `fr-CA`.

Как видите, выбор стандарта может быть не простым делом. Вы должны быть осторожны, когда делаете выбор, и уверены, что он действительно соответствует вашим потребностям.

Теперь, когда мы нашли правильный стандарт, можно переводить все свои сообщения об ошибках на те языки, которые мы хотим поддерживать. Например, при использовании цели «Перевод денег» с параметром `language`, установленным в `fr-FR`, сообщение об ошибке `AMOUNT_OVER_SAFE` может выглядеть так: `Le montant dépasse le seuil autorisé`. Обратите внимание, что любой текст, возвращаемый API, а не только сообщения об ошибках, может быть возвращен на языке, указанном в параметре `language`. Он также может быть представлен в качестве параметра запроса, но, поскольку банковский API – это REST API, вместо него можно использовать протокол HTTP.

**ВНИМАНИЕ!** Я не рекомендую использовать автоматический перевод; результат может быть далеко не точным и полностью разрушит ваши попытки понравиться потребителю и конечному пользователю.

Согласование содержимого относится не только к форматам данных, но и к языкам. Так же как потребители могут использовать HTTP-заголовки `Accept` и `Content-type` для указания медиатипа, они могут применять `Accept-Language` и `Content-Language`, чтобы указать, на каком языке говорят, как показано на рис. 6.10.

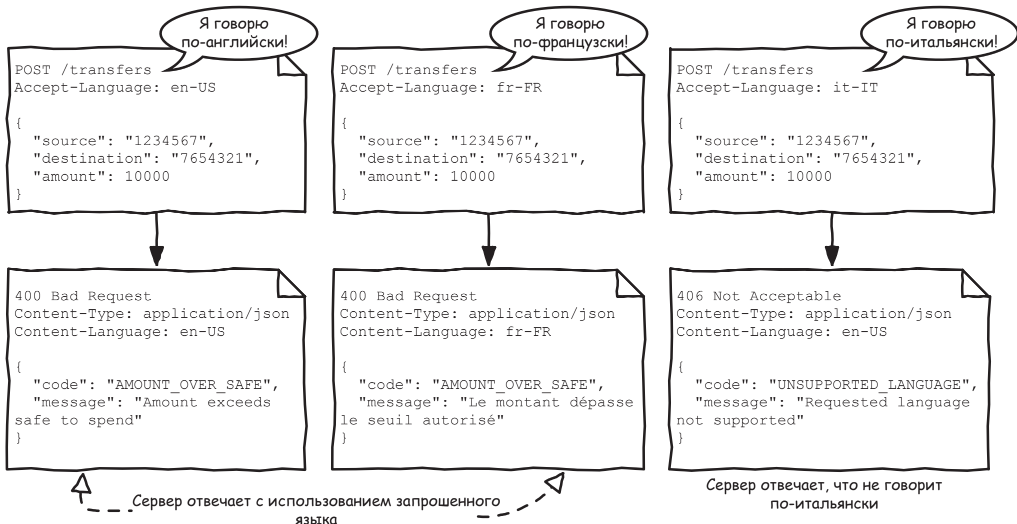



**Рис. 6.10. Согласование языка содержимого с API**

При использовании запроса `POST /transfer` для перевода денег, если потребители не предоставляют заголовки, сервер API может вернуть ответ с заголовком `Content-Language: en-US`, чтобы указать, что любое текстовое содержимое написано на американском английском. Однако, если потребители предоставляют HTTP-заголовок `Accept-Language: fr-FR` со своими запросами, чтобы указать, что они хотят получить содержимое на французском языке, сервер API отвечает заголовком `Content-Language: fr-FR`, и все текстовые данные будут переведены

на французский. Если запрашиваемый язык – например, итальянский (it-IT) – не поддерживается, сервер возвращает код состояния HTTP 406 Not Acceptable. Поскольку этот код состояния также может быть возвращен, когда потребитель запрашивает медиатип, который не поддерживается, рекомендуется также предоставить сообщение об ошибке с четким кодом ошибки, например UNSUPPORTED_LANGUAGE, и сообщением типа Requested language not supported.

Однако адаптация значений данных для разработчиков, их приложений и конечных пользователей – это не только языковой перевод. Например, в США люди используют имперскую систему измерения, а во Франции – метрическую. Люди в США и Франции не используют одинаковые единицы измерения, одинаковые форматы даты и числа или одинаковый формат бумаги. Возможность адаптации ко всем этим вариантам возможна, если ваш API поддерживает интернационализацию и локализацию (часто называемые i18n и l10n; номера указывают на количество символов между первой и последней буквой слова).

Для нашего банковского REST API *интернационализация* означает понимание того, что заголовок Accept-Language: fr-FR указывает, что потребитель хочет получить локализованный ответ с использованием французского языка и соглашений. На стороне сервера это означает, что, если запрашиваемая локализация поддерживается, содержимое будет возвращено в локализованном виде вместе с заголовком Content-Language: fr-FR. Если она не поддерживается, сервер возвращает код состояния 406 Not Acceptable.

Для банковского API *локализация* означает возможность обрабатывать локаль fr-FR. Возвращаемые данные должны быть на французском языке с использованием метрической системы, а документ PDF должен быть создан с использованием размера A4, а не формата, принятого, например, в США. Эта тема относится не только к API; данные вопросы применимы ко всем областям разработки программного обеспечения.

> **ПРИМЕЧАНИЕ.** Интернационализация (i18n) – это механизм, позволяющий программному обеспечению, приложению или API выполнять локализацию. Локализация (l10n) – это способность справляться с адаптациями к локали, которая в основном состоит из языка и региона или страны.

Но, будучи проектировщиками и поставщиками API, должны ли мы заботиться об интернационализации и локализации? Это совершенно законный вопрос, на который нужно рано или поздно ответить, когда вы будете проектировать API. Все зависит от характера вашего API и целевых потребителей и/или конечных пользователей. Если вам повезет, на данные, обмен которыми ведется через ваш API, проблемы локализации могут не повлиять вообще, поэтому, возможно, вам удастся обойти это. Если вы не нацелены на людей из разных регионов, интернационализация может и не потребоваться. Будьте осторожны, потому что иногда люди могут использовать разные локали в одной и той же стране (например, локали en-US или es-US в США).

Если вы думаете, что они вам не нужны, можете начать проектирование без функций интернационализации и позже обновить свой API, если это необходимо. Но имейте в виду, что добавление функций интернационализации в существующий API может быть легко и прозрачно выполнено с точки зрения потребителя. Модификация реализации, которая была создана без учета интернационализации, может быть более сложной.

Обратите внимание, что существуют и другие аспекты согласования содержимого, такие как приоритеты при запросе нескольких вариантов ресурса и кодировка содержимого, которые мы не будем рассматривать в этой книге. Вы можете прочитать об этом подробнее в RFC 7231 (https://tools.ietf.org/html/rfc7231).

Мы видели, что потребители могут указывать не только формат данных, который они хотят использовать, но также, например, язык и единицы измерения. Можно ли предоставить еще более настраиваемый API, чтобы он был еще более предсказуемым? Да, можно!

### 6.2.3 Фильтрация, разбиение на страницы и сортировка

Банковский счет, который был открыт в течение многих лет, может иметь тысячи транзакций. Клиент, который хочет получить транзакции по счету с помощью банковского API, вряд ли горит желанием увидеть все эти транзакции сразу и предпочитает получить определенное количество. Может быть, он хочет увидеть десять самых последних транзакций, а затем, возможно, пойти дальше по списку. Как показано на рис. 6.11, это можно сделать, добавив несколько необязательных параметров для этой цели, таких как `pageSize` и `page`.

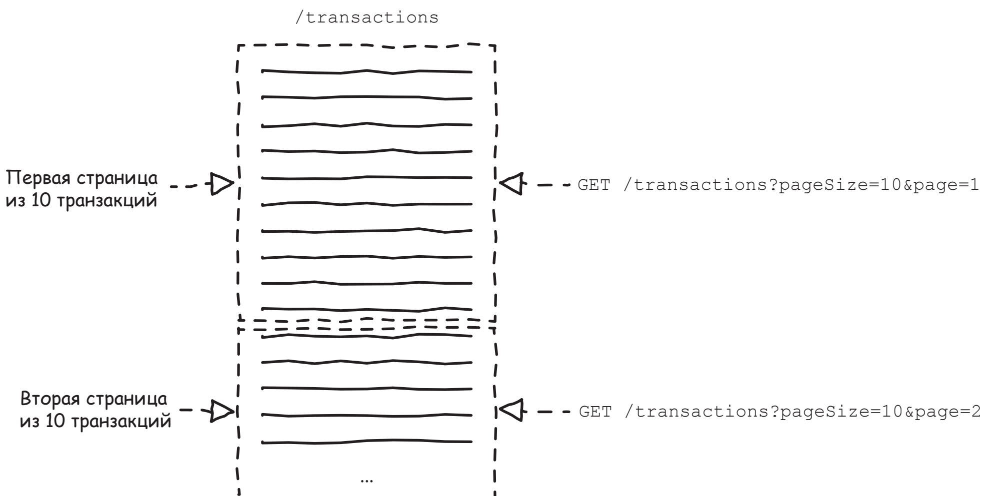



**Рис. 6.11. Простая нумерация страниц**

На сервере список транзакций фактически разделен на страницы, каждая из которых содержит транзакции `pageSize`. Если параметр `pageSize` не указан, сервер использует значение по умолчанию, а если не указан параметр `page`, сервер возвращает первую страницу по умол-

чанию. Чтобы получить первую страницу из десяти транзакций, потребители должны использовать это: pageSize=10 и page=1. Чтобы получить вторую страницу из десяти транзакций, необходимо использовать это: pageSize=10 и page=2.

В нашем банковском REST API эти параметры пагинации могут быть переданы как параметры запроса, например GET /account/1234567/transactions?PageSize=10&page=1. Но мы также могли бы воспользоваться преимуществами протокола HTTP и взять HTTP-заголовок Range. Чтобы получить первую страницу из десяти транзакций, этот заголовок должен быть Range: items = 0 - 9. Чтобы получить следующую страницу, заголовок должен выглядеть так: Range: items=10-19.

Заголовок Range был создан, для того чтобы веб-браузер мог получить часть двоичного файла. Значение заголовка запроса Range равно <unit>=<first>-<last>. Стандартная единица измерения – байты, поэтому код bytes=0-500 вернет первые 500 байт двоичного файла.

Мы можем использовать пользовательские единицы, такие как items. Отправка заголовка Range со значением items=10-19 сообщает серверу: «Я хочу, чтобы элементы коллекции шли от индексов с 10 по 19». Я мог бы выбрать другое имя, например transactions, но это означало бы, что, если бы нам нужно было разбить на страницы ресурс /accounts, единицей была бы accounts. Имя единицы, используемой для разбивки на страницы, можно угадать по названию коллекции, но я предпочитаю использовать общие имена. Таким образом, нет необходимости угадывать единицу для нумерации страниц: она всегда одна и та же.

Однако, если потребителям нужно несколько транзакций, им может потребоваться больший контроль над ними. Может быть, они хотят видеть только транзакции, которые были отнесены к категории *ресторанных* транзакций. Чтобы получить эти конкретные транзакции, потребитель может отправить запрос GET /account/1234567/transactions?category=restaurant. Параметр запроса category используется здесь для фильтрации транзакций и возврата только тех транзакций, которые относятся к категории «ресторан».

Этот пример фильтрации по сути является базовым. Если вы хотите попрактиковаться, вот проблема, которую вам рано или поздно придется решать, как проектировщику API: фильтрация коллекции по числовым значениям. Допустим, вы проектируете API для подержанных автомобилей. Пользователи должны иметь возможность увидеть список доступных автомобилей с пробегом между двумя значениями, используя запрос GET /cars и один или несколько параметров запроса. При использовании естественного языка такой запрос будет выглядеть примерно так: «Перечислить автомобили с пробегом от 15 000 до 30 000 миль». Попробуйте выполнить следующие упражнения:

- найдите способ проектирования такого фильтра;
- найдите по крайней мере два других способа сделать то же самое, выполнив поиск по существующим API (или руководствам по проектированию API);

- решите, какой вы предпочитаете;
- бонус: что касается других способов, опишите запрос и его параметр (параметры), используя спецификацию OpenAPI.

По умолчанию транзакции идут по порядку от самых последних до самых старых; когда потребители запрашивают транзакции, они сначала получают последние. Потребители могут также захотеть отсортировать транзакции в порядке убывания (сначала идут самые крупные суммы) и в хронологическом порядке (от самых старых до самых последних). Чтобы получить такой список, они могут отправить запрос:

```
GET /accounts/1234567/transactions?sort=-amount,+date
```

Параметр запроса `sort` определяет способ сортировки списка транзакций. Он содержит список пар типа «направление–свойство». Направление + (плюс) – по возрастающей и – (минус) – по нисходящей. Значения `-amount` и `+date` указывают серверу сортировать транзакции по сумме в порядке убывания и по дате в порядке возрастания. Обратите внимание, что это только один из способов предоставить параметров сортировки; это можно сделать и другими способами.

Эти функции разбивки на страницы, фильтрации и сортировки можно применять вместе. Использование параметров запроса `category=restaurant&sort=-amount,+date&page=3` в запросе `GET /accounts/1234567/transactions` возвращает третью страницу транзакций по ресторанам с указанием суммы в порядке убывания и дат в порядке возрастания.

Как показал этот раздел, помимо того, чтобы наш API выглядел знакомым, хороший способ сделать его предсказуемым – это позволить потребителям сказать, чего они хотят, и дать им это. Третий способ сделать API предсказуемым – предоставить потребителям подсказки касательно того, что они могут с ним сделать.

## 6.3 Быть видимым

В случае с большинством книг вы знаете, какую страницу вы читаете, потому что на ней напечатан ее номер. Иногда текущая глава или раздел также указывается в верхней или нижней части страницы. Ранее мы видели, что, читая какие-то книги, вы можете сразу перейти к определенной главе или разделу. Это возможно, потому что в книге есть удобное оглавление, где перечислены главы и разделы и на каких страницах они начинаются. Поэтому, когда вы работаете с книгой, вы читаете ее содержимое, но у вас также есть доступ к дополнительной информации о самом содержимом. Вы можете прочитать книгу, не используя эту информацию; если бы все это было удалено, содержимое не было бы затронуто. Но читать книгу было бы гораздо менее удобно.

Если книга – это роман, отсутствие оглавления (или даже номеров страниц) на самом деле не является проблемой. Роман интереснее читать постранично, не портя себе впечатление от слишком явного предварительного просмотра оглавления (например, «Глава 11: Персонаж,

к которому вы так привязались, умирает»). Но если это практическая книга, например та, которую вы сейчас читаете, можете вначале отсканировать оглавление, чтобы лучше понять, о чем эта книга, и быть уверенным, что она актуальна для вас. Вы также можете перейти к определенному разделу, потому что у вас есть конкретная проблема, которую нужно решить. Без оглавления и номеров страниц было бы нелегко найти то, что вам нужно. Эта дополнительная информация делает книгу *доступной для обнаружения*. Это необязательно, но значительно улучшает процесс чтения.

Как и книги, API-интерфейсы могут проектироваться, для того чтобы их можно было обнаружить. Это делается путем предоставления дополнительных данных различными способами, но обнаруживаемость также можно улучшить с помощью преимуществ используемого протокола. У REST API эта особенность заложена в генах, потому что они используют URL-адреса и протокол HTTP.

### 6.3.1 Предоставление метаданных

В разделе 6.2.3 вы познакомились с функцией разбиения на страницы. Получая доступ к списку транзакций по счету, пользователи банковского API могут указать, какая страница транзакций им нужна. Но как они узнают, что доступно несколько страниц?

На данный момент ответ сервера, когда потребители запрашивают список транзакций, состоит только из объекта, содержащего свойство `items`, которое представляет собой массив транзакций. Похоже на книгу без номеров страниц и оглавления. Этот ответ можно улучшить, добавив данные о разбиении на страницы, как показано на рис. 6.12.

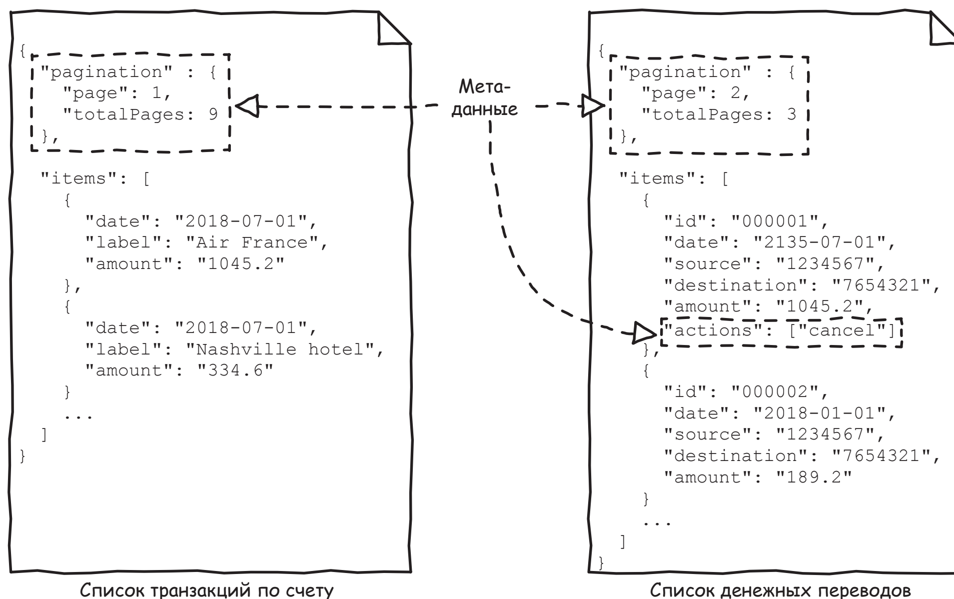



**Рис. 6.12.** Предоставление метаданных, чтобы объяснить, «где я и что я могу сделать»

Если первый вызов цели «Получить счет» выполняется без параметров разбиения на страницы, сервер может вернуть массив items вместе с номером текущей страницы (`page` со значением 1) и общим количеством страниц (`totalPages` со значением 9). Это скажет потребителям, что впереди еще восемь страниц транзакций.

Благодаря дополнительным данным список транзакций теперь доступен для обнаружения. В информатике такие данные называются *метаданными*; это данные о данных. Метаданные можно использовать, чтобы сообщить потребителям, где они находятся и что они могут сделать.

Давайте рассмотрим еще один пример, просто чтобы показать, что метаданные не ограничиваются нумерацией страниц. Используя банковский API, потребители могут переводить деньги с одного счета на другой сразу же или в заранее установленный день. При перечислении прошлых запросов на перевод сервер может возвращать как выполненные, так и отложенные запросы. Уже выполненный запрос нельзя отменить, но отложенный, который еще не был произведен, отменить можно. Как показано на рис. 6.12, мы можем добавить метаданные, описывающие возможные действия для каждого запроса на перевод. Для уже выполненного денежного перевода список действий будет пустым. В случае с отложенным переводов он может содержать элемент `cancel`. Это сообщит потребителям, какие из них они могут использовать, чтобы отменить перевод денег.

Как видите, API может возвращать метаданные вместе с данными, чтобы помочь потребителям узнать, где они находятся и что они могут сделать. API можно использовать и без этой дополнительной информации, но метаданные значительно облегчают его использование. Добавляя метаданные, мы в основном применяем знания, полученные в разделе 5.1, – предоставляем готовые данные.

Это можно делать с любым типом API. В зависимости от рассматриваемого API вы можете полагаться на другие механизмы для предоставления такой информации, особенно используя преимущества некоторых функций выбранного вами протокола.

### *6.3.2 Создание гипермедиа-API*

Используя банковский REST API, потребители могут получить список счетов с помощью запроса `GET /account`. Каждый счет имеет уникальный идентификатор, который можно использовать для создания своего URL-адреса (`/account/{accountId}`) и получить подробную информацию о нем, используя HTTP-метод `GET`. Этот идентификатор также можно использовать для получения транзакций с помощью `GET /accounts/{accountId}/transactions`. Благодаря метаданным пагинации, которые мы только что добавили, потребители узнают, будут ли еще транзакции, кроме тех, что были возвращены после первого вызова. В таком случае они могут использовать `GET /account/{accountId}/transactions?page=2`, чтобы получить следующую страницу транзакций. Они даже могут сразу перейти на последнюю страницу. Им просто нужно взять значение `lastPage`

и использовать его, чтобы выполнить запрос `GET /account/{accountId}/transactions?page={lastPage value}`.

Похоже на хорошо спроектированный API с кристально чистыми URL-адресами и даже метаданными, которые помогают потребителям, не так ли? Теперь давайте представим ситуацию, когда вы просматриваете сайт банка, а все гипермедиассылки были удалены. Если бы вам, как клиенту, пришлось прочитать руководство пользователя, чтобы узнать все доступные URL-адреса, вам бы это понравилось? Если вам нужно просмотреть подробную информацию об одном из ваших счетов и придется самостоятельно создавать URL-адрес страницы, копируя и вставляя номер счета, вам это понравится? Было бы ужасно использовать Всемирную паутину без гиперссылок.

К счастью, это работает не так. Попав на сайт, вы можете узнать его содержание, просто щелкнув по ссылкам, и перейти с одной страницы на другую. REST API основаны на принципах Всемирной паутины, так почему бы не воспользоваться ими? Как показано на рис. 6.13, банковский гипер-API будет предоставлять свойство `href` для каждого счета, возвращаемого с помощью запроса `GET /account`.

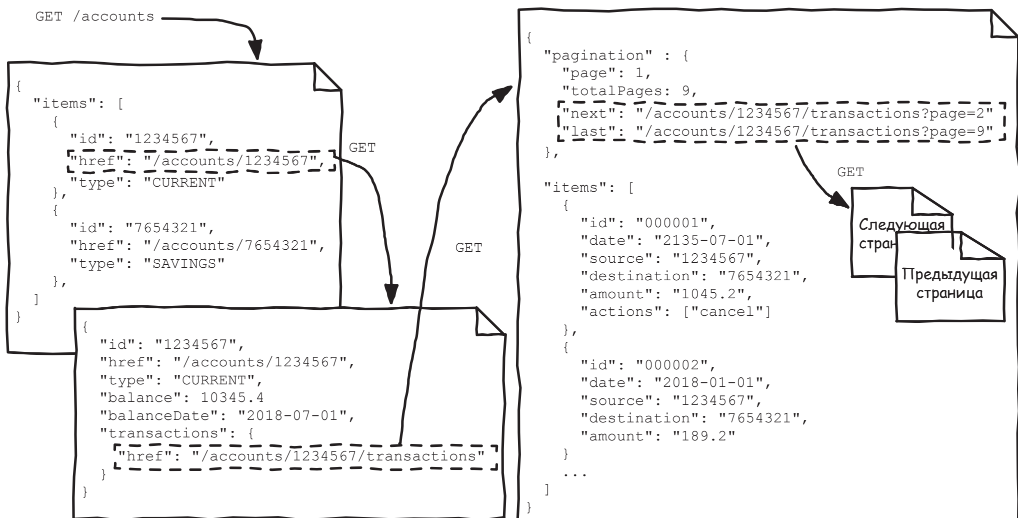



**Рис. 6.13.** Банковский гипер-API

В случае со счетом `1234567` его значением будет `/account/1234567`. Потребители, желающие получить доступ к подробной информации об этом счете, должны будут просто получить готовый к использованию относительный URL-адрес, не создавая его самостоятельно. А ответ на этот запрос будет иметь свойство `transactions`, значением которого может быть объект, содержащий свойство `href` со значением `/accounts/1234567/transactions`.

Опять же, потребители просто должны использовать метод `GET` и значение `href`, чтобы получить список транзакций. И конечно же, метаданные пагинации будут предоставлять URL-адреса, такие как `next` и `last`,

используя свойства, значения которых могут быть /account/1234567/transactions?page=2 и /accounts/1234567/transactions?page=9 соответственно. Тогда потребители смогли бы просматривать API без необходимости знать доступные URL-адреса и их структуры.

REST API предоставляют ссылки так же, как и веб-страницы. Это облегчает обнаружение API и, как вы увидите позже, обновление API. Стандартного способа предоставления этих гиперметаданных не существует, но есть общепринятые практики, по большей части основанные на том, как ссылки представлены на HTML-страницах и в протоколе HTTP.

В гиперметаданных обычно используются такие имена, как href, links или _links. Хотя здесь нет стандарта, было определено несколько форматов. Наиболее известные из них – HAL, Collection+JSON, JSON API, JSON-LD, Hydra и Siren. Эти форматы поставляются с различными ограничениями относительно структуры данных.

HAL (http://stateless.co/hal_specification.html) относительно простой формат. Базовый документ HAL имеет свойство links, содержащее доступные ссылки. Каждая ссылка – это объект, идентифицируемый связями (или _rel) с текущим ресурсом. Связь self используется для ссылки на ресурс. Объект ссылки содержит как минимум свойство href с полным или относительным URL-адресом. Для ресурса банковского счета ссылка на его транзакции будет находиться там как transactions, как показано в приведенном ниже листинге.

**Листинг 6.1. Банковский счет в виде документа HAL**

```
{
  "_links" : {
    "self": {
      "href": "/accounts/1234567" ①
    },
    "transactions": {
      "href":"/accounts/1234567/transactions" ②
    }
  }
  "id": "1234567",
  "type": "CURRENT",
  "balance": 10345.4
  "balanceDate": "2018-07-01"
}
```

① Ссылка на сам ресурс банковского счета.

② Ссылка на транзакции банковского счета.

Концепция ссылочных связей не является специфичной для HAL; она определена в RFC 5988 (https://tools.ietf.org/html/rfc5988). Гипер-API не только предоставляют доступные URL-адреса; они также могут предоставлять доступные методы HTTP. Например, в формате Siren (https://github.com/kevinswiber/siren) можно описать действие cancel при отложенном денежном переводе. У Siren также есть ограничения относительно

но структуры данных: свойства группируются в свойства (`properties`), ссылки на другие ресурсы находятся в `links`, а действия – в `actions`. В приведенном ниже листинге приводится пример документа Siren.

**Листинг 6.2. Денежный перевод в виде документа Siren**

```
{
   "properties" : { ①
      "id": "000001",
      "date": "2135-07-01",
      "source": "1234567",
      "destination": "7654321",
      "amount": "1045.2"
   },
   links: [ ②
      { "rel": ["self"],
         "href": "/transfers/000001" }
   ],
   actions: [ ③
      { "name": "cancel",
         "href": "/transfers/000001",
         "method": "DELETE" }
   ],
}
```

① **Группирует свойства ресурса по свойствам.**
② **Эквивалент _links в HAL.**
③ **Описывает действие, используя имя, URL-адрес и метод HTTP.**

## Ограничения REST: единообразие интерфейса

Архитектурный стиль REST гласит, что все взаимодействия должны руководствоваться концепцией идентифицированных ресурсов, которыми манипулируют посредством представлений состояний ресурсов и стандартных методов, и предоставляет все метаданные, необходимые для понимания представлений и знания того, что можно сделать с этими ресурсами.[1] API-интерфейсы REST – это гипермедиа-API, которые предоставляют все метаданные, необходимые, для того чтобы потребители могли путешествовать по ним, как по сайту, чтобы облегчить их обнаружение. Метаданные могут использоваться для описания не только связей между ресурсами, но и между доступными операциями. Эта часть ограничения унифицированного интерфейса архитектурного стиля REST носит название *Hypermedia as the Engine of Application State* (часто используется вариант в виде непроизносимой аббревиатуры HATEOAS).

[1] *Рой Томас Филдинг*, Архитектурные стили и проектирование сетевых программных архитектур, 2000 (https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2).

Предоставление метаданных гипермедиа является наиболее распространенным способом использования веб-корней REST API для созда-

ния предсказуемых API, но протокол HTTP предоставляет функции, которые можно использовать, для того чтобы сделать REST API еще более предсказуемыми.

### 6.3.3 Использование преимуществ протокола HTTP

До сих пор мы применяли HTTP-методы `GET`, `POST`, `PUT`, `DELETE` и `PATCH`. В приведенном ниже листинге показано, что запрос `OPTIONS /transfers/000001` можно использовать для определения доступных методов HTTP в ресурсе.

**Листинг 6.3. Использование HTTP-метода OPTIONS**

```
OPTIONS /transfers/000001

200 OK
Allow: GET, DELETE
```

Если сервер API поддерживает этот метод и ресурс существует, он может вернуть ответ `200 OK` вместе с заголовком `Allow: GET, DELETE`. В ответе четко указано, что HTTP-методы `GET` и `DELETE` можно использовать в `/transfers/000001`. Подобно метаданным, которые могут предоставить информацию о данных (раздел 6.3.1), такие метацели могут предоставлять информацию о целях API.

Ранее в этой главе вы видели, что список транзакций по счету можно вернуть в виде документа JSON, CSV или PDF. В приведенном ниже листинге показано, что при ответе на запрос `GET /accounts/1234567/transactions` сервер API может указать другие доступные форматы с помощью заголовка `Link`.

**Листинг 6.4. Ответ, указывающий на другие доступные форматы с помощью заголовка Link**

```
200 OK
Allow: GET
Content-type: application/json <---
Link:</accounts/1234567/transactions>;
          type=application/pdf,
        </accounts/1234567/transactions>; <---
          type=text/csv
   {
     "items" : [
        ...
     ]
   }
```

① **Список транзакций возвращается в виде документа JSON.**
② **Он также доступен в форматах PDF и CSV (обратите внимание, что на самом деле это одна строка).**

Обратите внимание, что такое использование протокола HTTP API-интерфейсами REST применимо не везде. Как и при выборе стан-

дарта, нужно проверить, действительно ли такие функции полезны для потребителей. Если это так, вам, возможно, придется подробно объяснить их в своей документации тем, кто не является экспертами по протоколу HTTP.

**СОВЕТ.** Всегда проверяйте возможности протокола, используемого вашим API, но будьте осторожны, чтобы не запутать пользователей редко используемыми функциями. Вы можете использовать их, но они должны быть тщательно задокументированы.

Если хотите попрактиковаться в уроках из этого раздела, можете попробовать обновить Shopping API, с которым мы работали в главах 3 и 4, используя спецификацию OpenAPI следующим образом:

- добавьте функции гипермедиа, используя HAL (или Siren) для представления ссылок между ресурсами;
- добавьте функции разбиения на страницы, фильтрации и сортировки с соответствующими метаданными и элементами управления гипермедиа;
- добавьте функцию согласования содержимого для поддержки формата CSV;
- при необходимости добавьте HTTP-метод `OPTIONS`.

В следующей и последней главе, посвященной юзабилити, вы научитесь организовывать и масштабировать свои API-интерфейсы, чтобы они оставались удобными в использовании.

## *Резюме*

- Чтобы создавать API, чьи действия можно угадать, последовательно определяйте соглашения и следуйте общепринятым практикам и стандартам.
- Согласованность в вашем дизайне упрощает не только использование API, но и его проектирование.
- Всегда проверяйте, нужно ли давать вашему API другое представление и/или возможности для локализации и интернационализации.
- Для каждой цели, связанной со списками, подумайте, можно ли облегчить ее использование с помощью пагинации, фильтрации и сортировки.
- Чтобы направлять потребителей, предоставьте как можно больше метаданных (например, гиперссылок).
- Всегда проверяйте базовый протокол и используйте его доступные возможности, чтобы сделать ваш API предсказуемым, стараясь не путать пользователей сложными или полностью неиспользуемыми функциями.

7

# Проектирование лаконичного и хорошо организованного API

**В этой главе мы рассмотрим:**

- организацию данных API, ответных сообщений и целей;
- управление детализацией данных, целей и API.

Теперь, когда вы знаете, как проектировать простые и предсказуемые API-интерфейсы, нужно рассмотреть еще кое-что, чтобы убедиться, что мы проектируем *удобные для использования* API. Телевизионные пульты с их многочисленными и не всегда хорошо организованными кнопками иногда выглядят пугающе. Некоторые микроволновые печи или стиральные машины предлагают слишком много функций для простых смертных. Перегруженные, неорганизованные, непонятные или пестрые повседневные интерфейсы в лучшем случае озадачивают своих пользователей, а в худшем – пугают их.

«Меньше значит больше» и «все и вся на своем месте» – два афоризма, которые должен применять каждый проектировщик API. Организация и определение размера данных API, ответных сообщений и целей важны для обеспечения API, который легко понять и который не будет перегружать пользователей. В противном случае все, что мы узнали о создании простых и предсказуемых API, ничего не стоит.

## 7.1 Организация API

Если вы когда-либо пользовались пультом для телевизора, то должны понимать значение всех кнопок из четырех примеров, показанных на рис. 7.1. Все они предлагают одинаковые функции с помощью 15 кно-

пок; но, в зависимости от организации кнопок, удобство пользования ими меняется от ужасного до идеального.

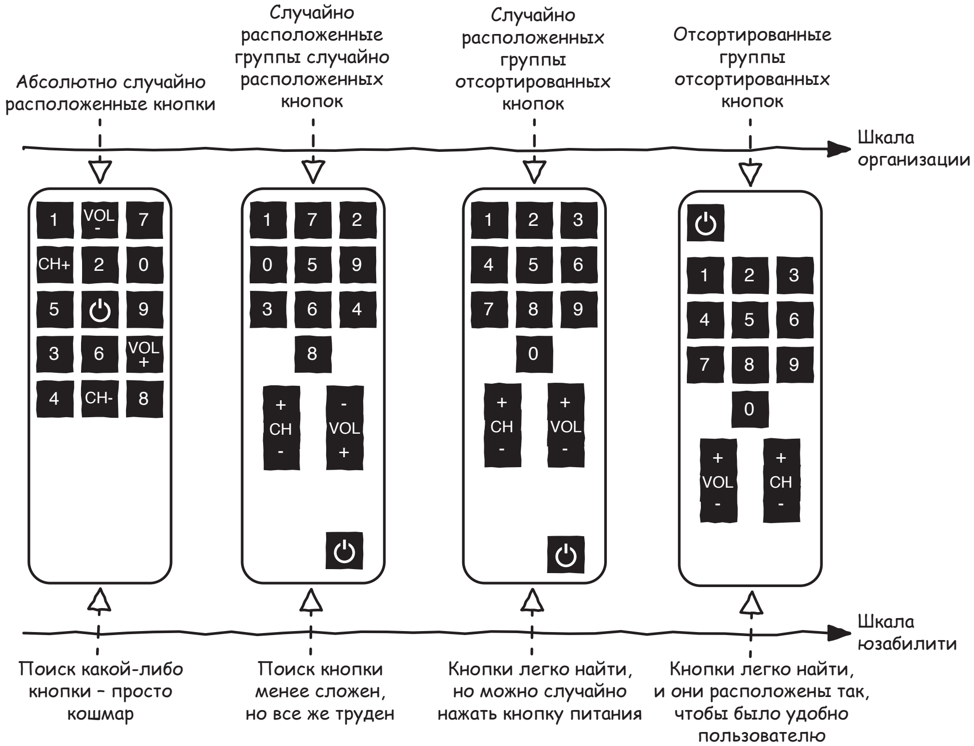



**Рис. 7.1.** Организация кнопок пульта дистанционного управления телевизора влияет на удобство использования

На первом пульте дистанционного управления (слева) кнопки расположены случайным образом, что затрудняет их поиск. На втором пульте кнопки сгруппированы по типу, что облегчает их поиск. Например, если пользователи ищут 7-ю кнопку, они знают, что могут найти ее в группе кнопок с цифрами. Кнопки переключения каналов (**CH**) и регулировки громкости (**VOL**) также сгруппированы вместе.

Но одной группировки недостаточно. На третьем пульте кнопки в каждой группе отсортированы. Теперь 7-ю кнопку легче найти благодаря сортировке по возрастанию в группе номеров. Поместить кнопку громкости поверх своей группы также лучше по двум причинам:

- значки +/– обычно располагаются таким образом, чтобы имитировать то, что они обозначают: вверх/вниз;
- кнопка **канал+** находится поверх своей группы, поэтому это изменение означает, что две группы являются согласованными.

Наконец, кнопки на четвертом пульте переставлены, чтобы пользоваться им было еще проще. Кнопка включения расположена сверху, чтобы предотвратить ее непреднамеренное нажатие, когда пользователь

держит пульт дистанционного управления. Кнопки переключения каналов и регулировки громкости также используются в соответствии с общепринятой практикой. Теперь, когда кнопки расположены интуитивно, вы можете с легкостью использовать пульт – возможно, даже в темноте во время просмотра фильма.

Как показывает этот пример, найти конкретный элемент в наборе и понять его назначение проще, если элементы сгруппированы и отсортированы логически. То же самое относится и к API – важна организация. Как и повседневный объект, API может быть либо непригодным для использования, либо совершенно интуитивно понятным, в зависимости от организации его данных, ответных сообщений и целей.

### 7.1.1 Организация данных

Проектирование хорошо организованного API начинается с данных, поэтому давайте рассмотрим конкретный пример. В изначальном представлении банковского счета, показанном на рис. 7.2 (a), выделено

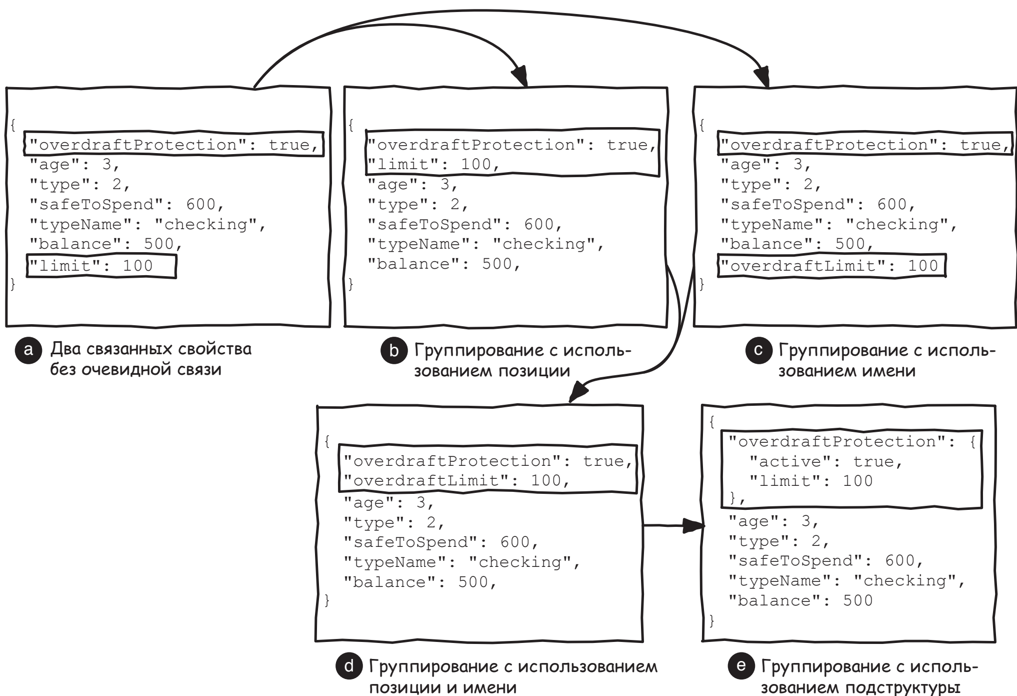



Рис. 7.2. Группировка данных в представлении банковского счета

два свойства: `overdraftProtection` и `limit`. Свойство `overdraftProtection` указывает на то, активна ли защита от овердрафта, а свойство `limit` указывает на то, насколько доступна защита от овердрафта. Здесь его значение равно 100, а это указывает на то, что любая транзакция, в результате которой баланс счета превысит 100 долл., будет заблокирована. Эти два свойства связаны, но в дизайне ничего явно не говорит нам об этом.

Мы можем внести изменения, чтобы сделать эту связь более очевидной. Сперва можно было бы поместить два свойства ближе друг к другу (b), но до сих пор не очевидно, что они связаны. Переименование свойства `limit` в `overdraftLimit` дает более подходящий результат (c), но сочетание этих двух методов все же лучше (d). Таким образом, мы создаем виртуальную границу вокруг них. Также можно создать более сплошную границу, поместив два эти свойства в подструктуру `overdraftProtection` (e).

Группировка данных – это первый шаг, но этого недостаточно. Необходимость постоянно прокручивать ответы API, чтобы найти наиболее важные данные, может ужасно раздражать. Сортировка данных может улучшить восприятие для людей (программы это вообще не заботит). Как показано на рис. 7.3, две другие группы можно создать, если поместить свойства `type` и `typeName`, а также `safeToSpend` и `balance` поближе друг к другу. В каждой группе свойства сортируются от более важных (вверху) к менее важным (внизу). Также все группы отсортированы по значимости.

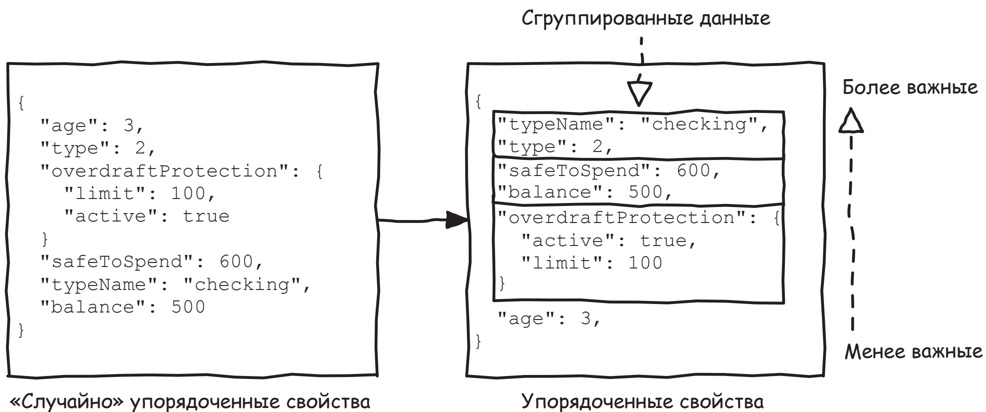



**Рис. 7.3. Сортировка данных в представлении банковского счета**

Такая организация также будет видна в документации или коде, сгенерированном из спецификации вашего API. Группировка свойств в выделенной структуре также может помочь обеспечить лучшее видение того, что нужно или не нужно, как показано на рис. 7.4.

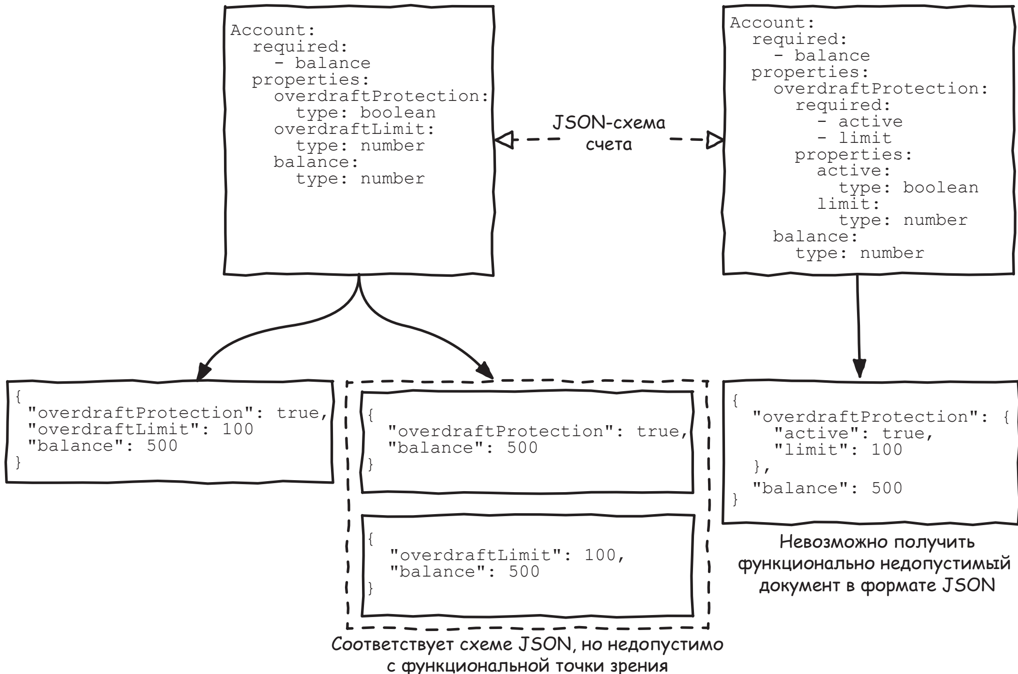



**Рис. 7.4.** Группировка данных для управления необязательной группой обязательных свойств

В этом примере, согласно JSON Schema слева, свойства `overdraftProtection` и `overdraftLimit` являются необязательными. Но с функциональной точки зрения если `overdraftProtection` имеет значение `true`, тогда свойство `overdraftLimit` является обязательным. Группировка этих двух свойств в необязательном объекте `overdraftProtection`, содержащем два обязательных свойства `active` и `limit`, решает эту проблему.

Не то чтобы такая стратегия в основном представляла точку зрения поставщика – здесь ограничение JSON Schema, которое не позволяет описывать сочетания обязательных и необязательных свойств. Но всегда полезно знать этот трюк; иногда это может здорово помочь, чтобы предоставить очень точную JSON-схему.

Чтобы спроектировать удобные в использовании данные, нужно организовать их путем создания групп данных – помещая связанные свойства ближе друг к другу, используя общепринятые префиксы или создавая подструктуры – и сортировки данных в этих группах и самих групп от более важных к менее важным.

### *7.1.2 Организация ответных сообщений*

Хорошо организованный API обеспечивает хорошо организованную обратную связь. В разделе 5.2 вы узнали, как можно использовать коды состояния HTTP для обеспечения информативных ответных сообще-

ний. Напоминаем, что в табл. 7.1 показаны некоторые из рассмотренных вами вариантов применения.

**Таблица 7.1. Примеры кода состояния HTTP**

| Случай использования | Код состояния HTTP | Класс | Значение |
|---|---|---|---|
| Создание денежного перевода | `201 Created` | 2XX | Создается мгновенный денежный перевод |
| Создание денежного перевода | `202 Accepted` | 2XX | Создается отложенный денежный перевод |
| Создание денежного перевода | `400 Bad Request` | 4XX | Отсутствует обязательное свойство или тип данных неверен |
| Получение банковского счета | `200 OK` | 2XX | Возвращается запрошенный банковский счет |
| Получение банковского счета | `404 Not Found` | 4XX | Запрашиваемый банковский счет не существует |

Коды состояния HTTP сгруппированы в *классы*. Ответ в классе 2XX означает, что все прошло нормально, а ответ 4XX – что с запросом есть проблема и потребитель должен ее исправить. Группирование кодов состояния HTTP таким способом облегчает их понимание.

Если API в ответ на ваш запрос возвращает код состояния 413, вы знаете, что проблема на вашей стороне, даже если вы никогда раньше не видели этот код[1]. Почему? Потому что это код класса 4XX. Он должен восприниматься так же, как и код состояния 400 (см. раздел 5.2.3)[2]. Однако организация ответных сообщений касается не только кодов состояния. Вы также можете организовать более конкретные или индивидуальные ответы. В разделе 5.2.4 вы увидели, что при возврате нескольких ошибок может быть полезно их классифицировать. На рис. 7.5 показан ответ на запрос на создание отложенного денежного перевода, в котором указана сумма, превышающая безопасный лимит. Здесь отсутствует целевой счет, а дата исполнения указана в виде временной метки UNIX.

[1] Этот код фактически означает, что ваш запрос больше, чем сервер хочет или может обработать (https://tools.ietf.org/html/rfc7231#section-6.5.11).

[2] RFC7231 утверждает, что «...клиент ДОЛЖЕН понимать класс любого кода состояния, как указано в первой цифре, и обрабатывать нераспознанный код состояния как эквивалентный коду состояния x00 этого класса...» (https://tools.ietf.org/html/rfc7231#section-6).

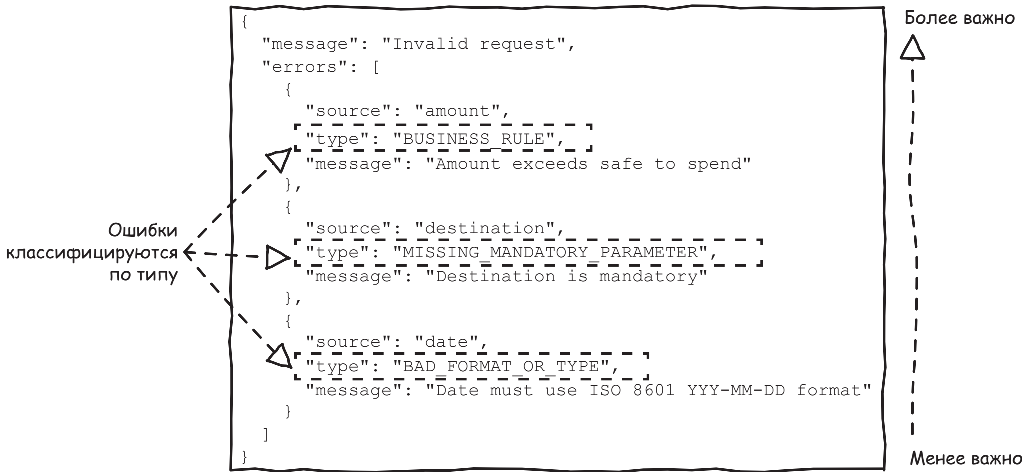



**Рис. 7.5.** Сгруппированные и отсортированные ошибки

Как видите, у каждой ошибки есть тип, который дает нам подсказку об источнике проблемы. Первая ошибка из группы `BUSINESS_RULE` явно касается вопросов контроля. Ошибка `MISSING_MANDATORY_PARAMETER`, очевидно, касается отсутствующего обязательного параметра. А очевидная ошибка `BAD_FORMAT_OR_TYPE` говорит нам, что значение свойства не соответствует ожидаемому типу или формату. Ошибки также сортируются от наиболее критическим к наименее критическим: `Amount exceeds safe to spend` – самая серьезная ошибка, за которой следуют `Destination is mandatory` и наименее критичная `Date must use ISO 8601 YYY-MM-DD format`.

При проектировании API вы должны организовать ответные сообщения, чтобы облегчить интерпретацию, используя преимущества обратной связи базового протокола, создавая собственную организацию ответных сообщений и сортируя ошибки от наиболее важных к наименее критическим.

### 7.1.3 Организация целей

Наконец, что не менее важно, цели также заслуживают того, чтобы быть хорошо организованными. Если вы знакомы с объектно-ориентированным программированием, то это можно сравнить с организацией методов в классах. Цели API могут быть организованы как виртуально, так и физически. Как показано на рис. 7.6, спецификацию OpenAPI, с которой вы познакомились в главе 4, можно использовать для *виртуальной* организации целей API.

Слева на рис. 7.6 приведено полностью дезорганизованное определение банковского API. Справа цели были сгруппированы в две категории, `Account` и `Transfer`, путем добавления свойства `tags` к каждой операции.

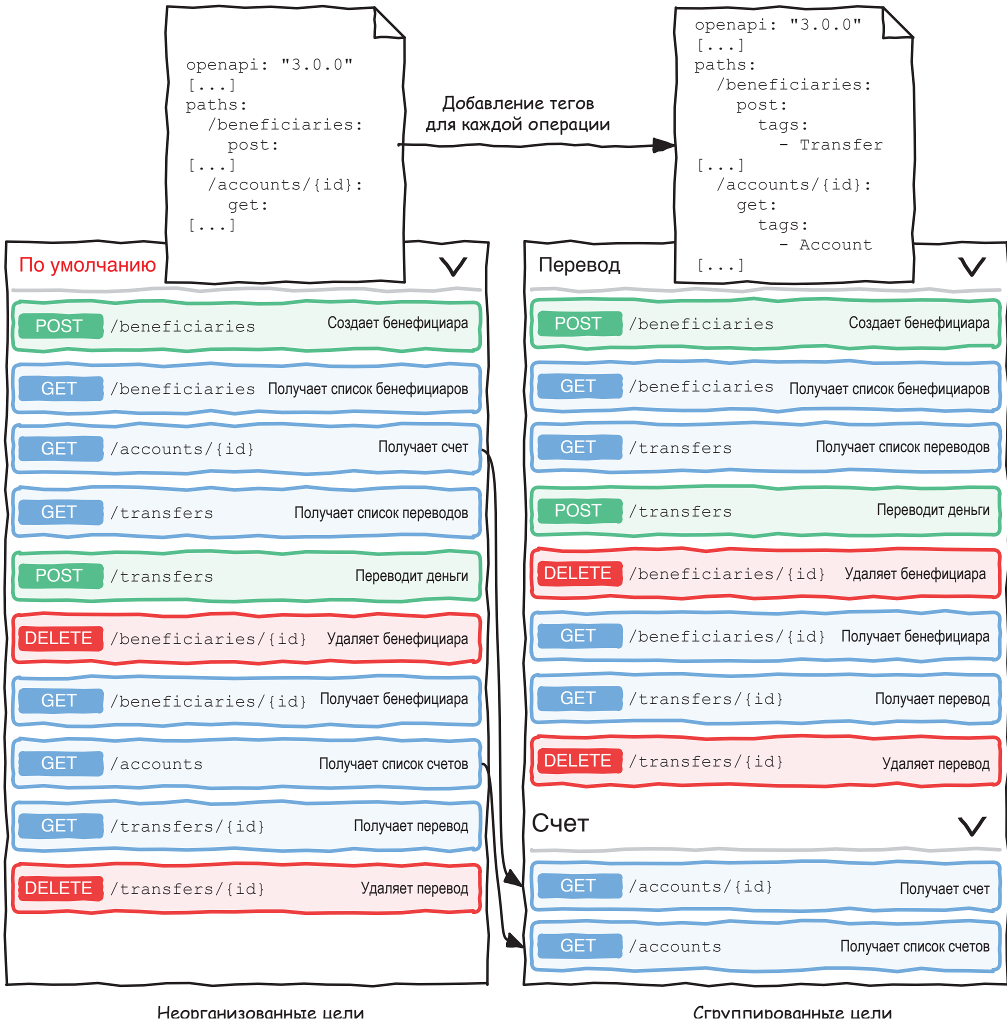



**Рис. 7.6.** Группировка целей с тегами в документе OpenAPI

**ПРИМЕЧАНИЕ.** При необходимости операция может принадлежать нескольким категориям.

Но как мы выбрали группу каждой цели? Волшебного рецепта нет, но идея состоит в том, чтобы сгруппировать цели, которые связаны с функциональной точки зрения. Если вы проектируете REST API, вас не должны сбивать с толку пути; нужно сосредоточиться на функциональности целей, а не на их представлениях, как показано на рис. 7.7.

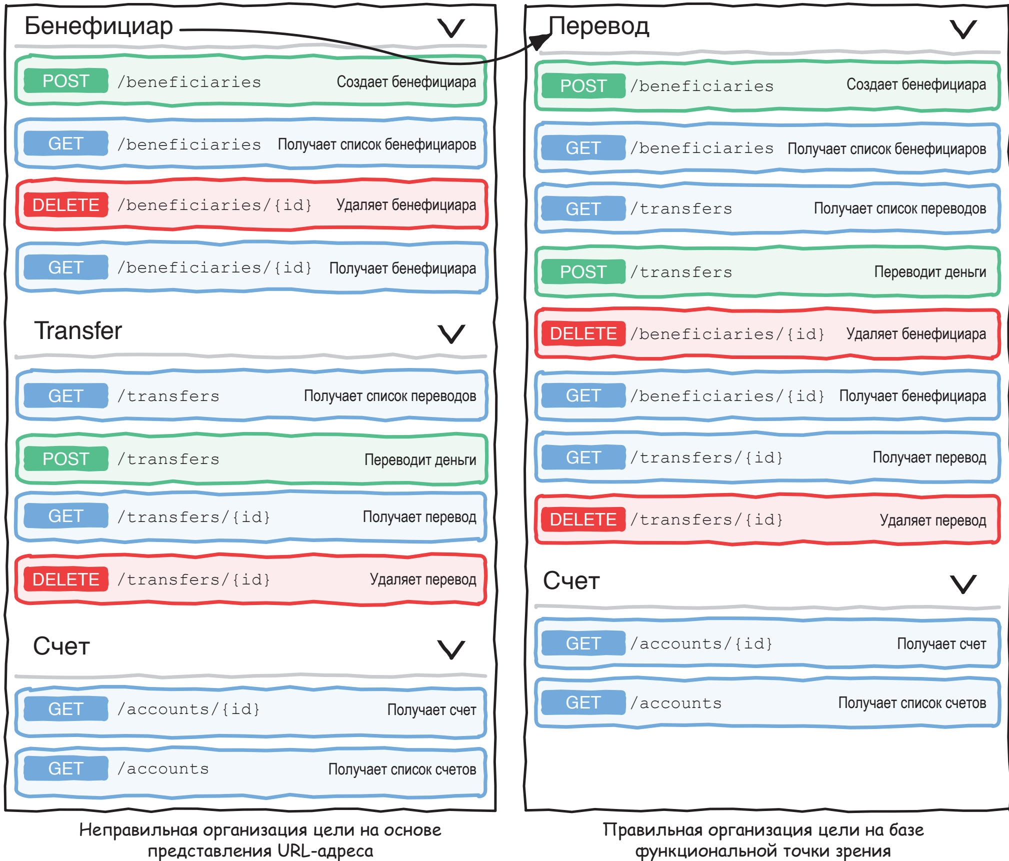



**Рис. 7.7.** Группировка целей с тегами на базе функциональности в сравнении с представлением URL-адреса

Как видно слева на этом рисунке, если бы мы сосредоточились на путях, то у нас бы получилось три категории: `Beneficiary` (для путей `/beneficiaries`), `Transfer` (для `/transfers`) и `Account` (`/accounts`). Но не имеет смысла разделять цели, представленные путями `/transfers` и `/beneficiaries`, потому что они не могут существовать друг без друга.

Справа мы разбили цели на две категории. Так лучше, но наличие категории `Transfer` перед `Account` не отражает то, как люди будут использовать API. Пользователи, скорее всего, сначала будут интересоваться операциями, связанными со счетами, прежде чем пытаться использовать API для перевода денег. Как показано на рис. 7.8, можно отсортировать категории, добавив определение `tags` на корневом уровне. Свойство `tags` – это список, в котором каждый элемент содержит имя тега и его описание.

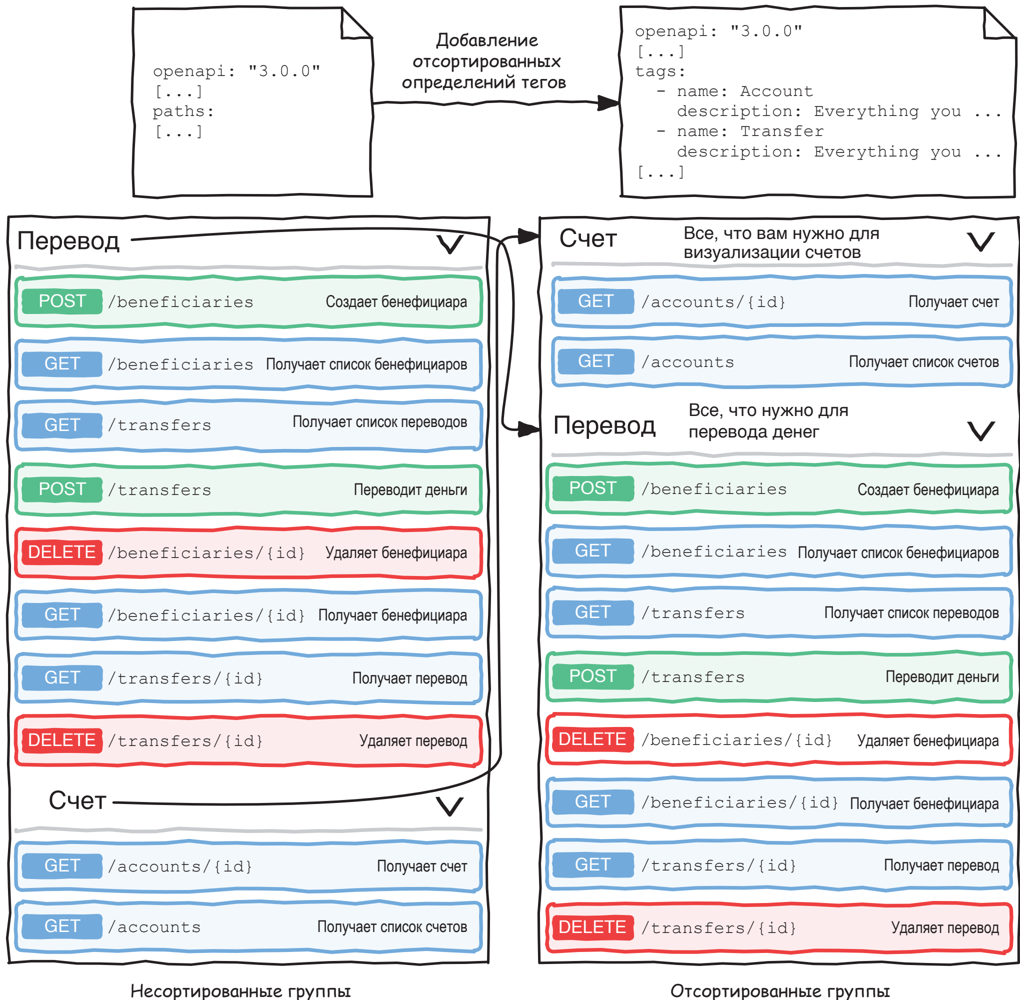



**Рис. 7.8.** Сортировка групп целей путем добавления определений отсортированных тегов

Все, что нам нужно, чтобы отсортировать теги `Account` и `Transfer`, как мы хотим, – это отсортировать определения тегов в списке `tags`. Затем можем добавить описание для каждого тега (или каждой категории). Теперь, когда группы отсортированы, следует также отсортировать операции внутри групп, как показано на рис. 7.9.

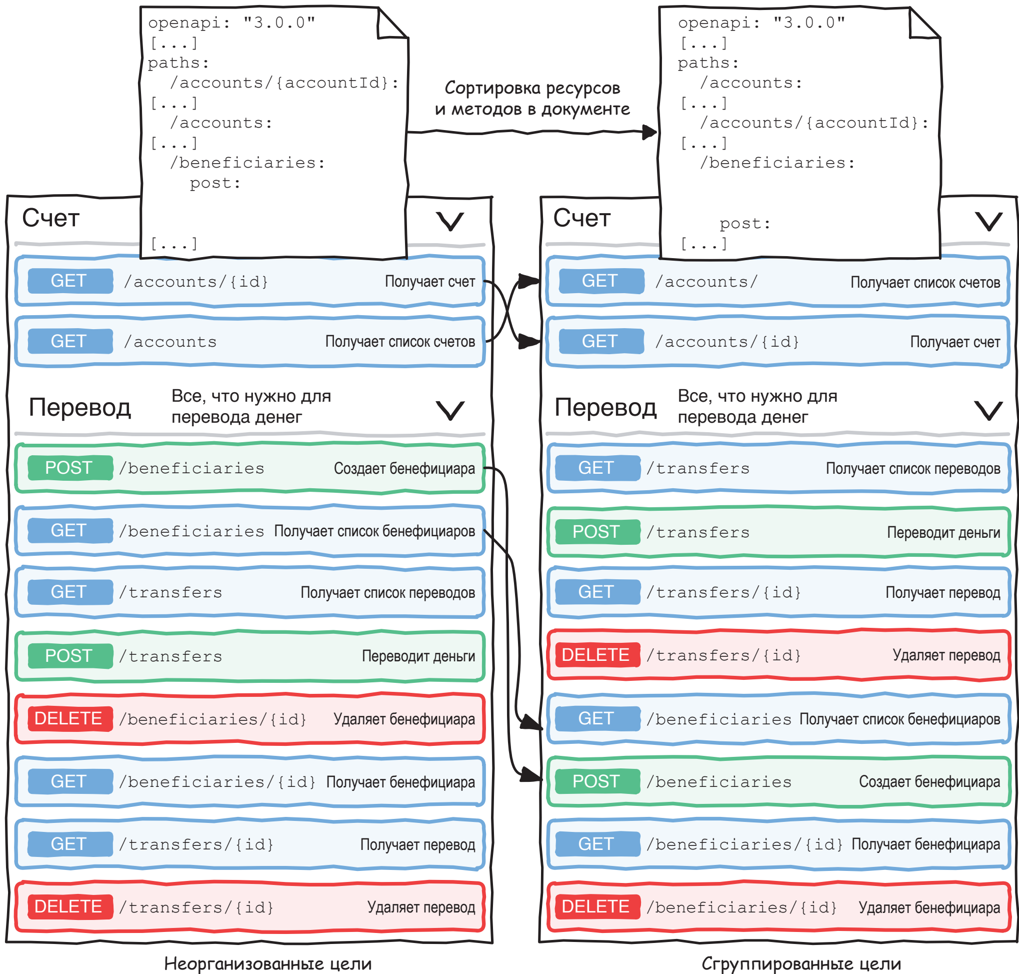



**Рис. 7.9. Сортировка целей в документе OpenAPI**

Это можно сделать только путем сортировки целей по порядку важности в самом документе спецификации (кстати, так и должно было быть сделано с самого начала). Здесь цель `GET /accounts` более важна, нежели `GET /accounts/{id}`, потому что потребители обычно перечисляют счета перед доступом к подробной информации о счете. Также обратите внимание на то, что порядок HTTP-методов одинаков для всех ресурсов: `GET`, `POST`, `DELETE`; `POST` и `GET` в `/beneficiaries` поменялись местами. Помните, что вы узнали о согласованности в разделе 6.1? Выберите один из способов сортировки HTTP-методов и придерживайтесь его в отношении всех ресурсов! Но это только *виртуальная* организация, которая не представлена в самом дизайне API. Мы могли бы добавить корневые пути `/account` и `/transfer` в пути ресурсов для их фактической группировки, как показано на рис. 7.10.

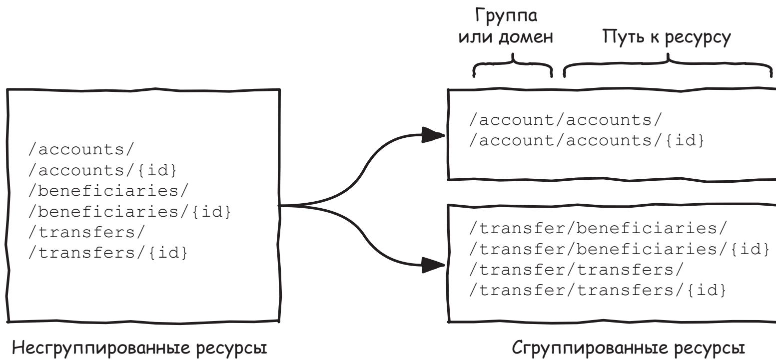



**Рис. 7.10. Группировка ресурсов по путям**

Тогда пользователи смогут устанавливать соединения между ресурсами, просто просматривая их пути. Однако имейте в виду, что из-за этого в некоторых случаях пути будет сложнее угадывать.

Теперь ваша очередь. Предположим, что цели, показанные на рис. 7.11, принадлежат API известной социальной сети обмена изображениями под названием Imagebook. Как бы вы организовали их, используя полученные знания?

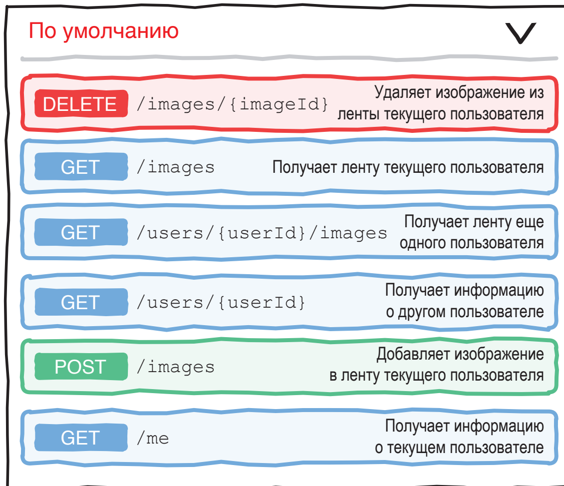



**Рис. 7.11. Как бы вы организовали эти цели?**

Организация целей API виртуально или физически облегчает понимание. Это можно сделать, отсортировав цели в документе определения и воспользовавшись форматом спецификации API. И вы можете добавить уровень организации при проектировании программного интерфейса (например, добавить уровень в путь для метода REST HTTP).

Теперь мы знаем, как спроектировать хорошо организованный API. Но, будучи проектировщиками API, мы должны убедиться, что наши API также являются *лаконичными*.

## 7.2 Определение размера API

В фильме Джо Данте 1984 года «Гремлины» Рэндалл Пельтцер, отец главного героя, пытается продать «изобретение века»: дружка для ванной (рис. 7.12).

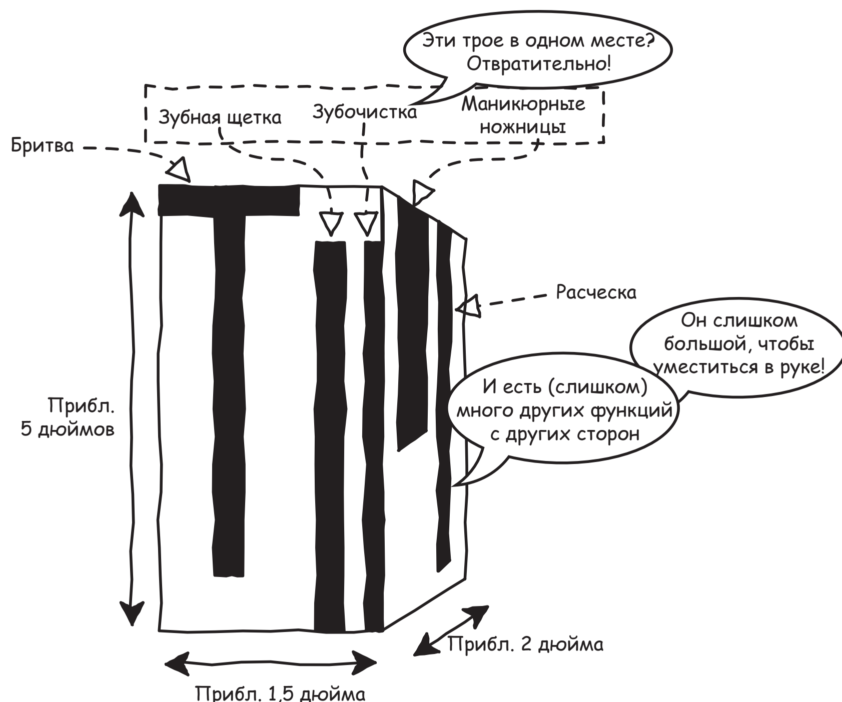



**Рис. 7.12.** Не очень удобный (и немного примитивный) дружок для ванной, который делает слишком много вещей

Это универсальное устройство для путешественников. Представьте себе предмет, похожий на огромный швейцарский армейский нож, включающий в себя бритву, дозатор крема для бритья, зеркало для бритья, зубную щетку, дозатор зубной пасты, зубочистку, зубное зеркало, расческу, кусачки для ногтей и, возможно, некоторые другие более или менее полезные аксессуары.

Проблема с этим устройством не в том, что каждая демонстрация с треском проваливается, в результате чего изобретатель сидит весь в зубной пасте или креме для бритья. Реальная проблема заключается в том, что это устройство хочет делать слишком много вещей, а это не очень удобно. Оно слишком большое, чтобы уместиться в руке, и использование каждого аксессуара кажется довольно сложной задачей. В первые несколько раз найти место, где спрятана расческа, может быть нелегко, а идея использовать одно и то же устройство для чистки зубов и обрезания ногтей на ногах по меньшей мере отвратительна.

Гораздо удобнее (и привлекательнее) использовать зубную щетку и кусачки для ногтей по отдельности друг от друга!

**ПРИМЕЧАНИЕ.** Объекты, предоставляющие слишком большое количество функций, чрезвычайно много элементов управления или чересчур много информации, обычно не используются. Обычно они бывают громоздкими, неудобными и пугающими.

Размер имеет значение не только для повседневных предметов. Каков правильный размер таблицы базы данных? Класс? Метод? Функция? Приложение? Это вопросы, которые постоянно возникают, когда вы работаете с программным обеспечением, и API не исключение. Каждый аспект API, в том числе его данные и цели, следует оценивать с умом. Иногда можно обнаружить, что нечто, рассматриваемое вами как единый API-интерфейс, стоить разделить, как в случае с зубной щеткой и кусачками для ногтей в дружке для ванной.

### 7.2.1 Выбор детализации данных

Представление банковского счета в формате JSON на рис. 7.13 содержит 32 свойства и имеет максимальную глубину 4.

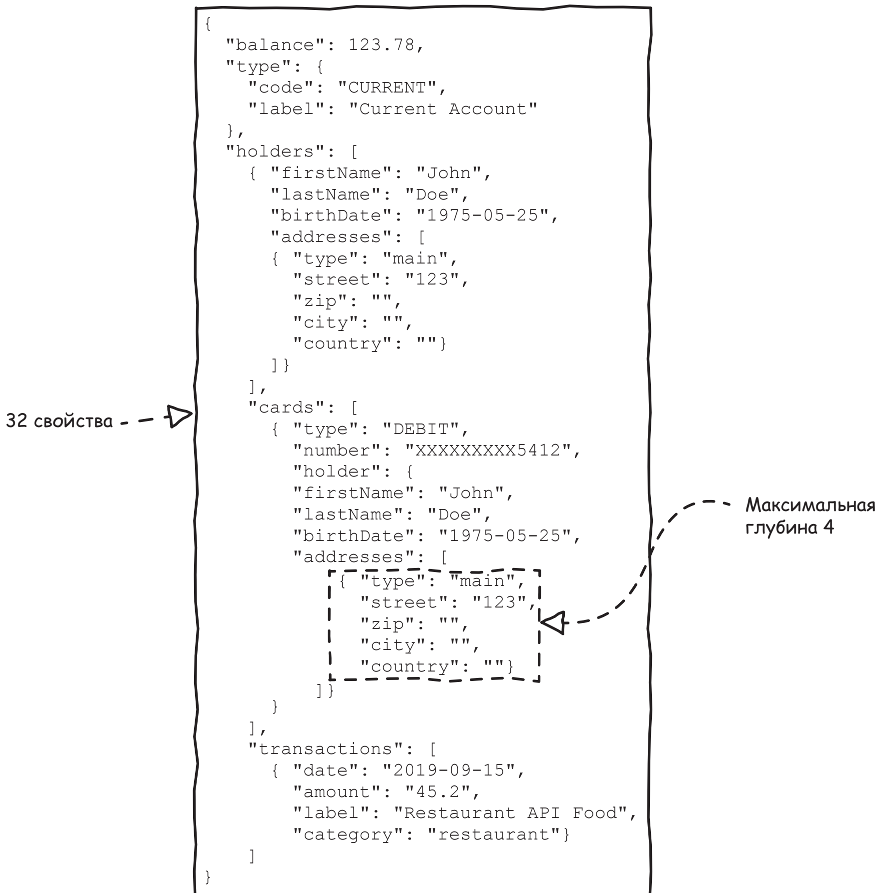



Рис. 7.13. Количество свойств и максимальная глубина представления банковского счета

Наличие 32 свойств кажется разумным; но, в зависимости от контекста, это может быть слишком много. Что, если это представление используется в списке? В этом случае может быть нецелесообразно предоставлять *всю* информацию о счете, когда пользователям, возможно, нужна только сводка. Также, если приглядеться, можно увидеть по крайней мере одну потенциальную проблему: этот банковский счет содержит список транзакций. Возможно, это представление пытается сделать слишком много всего одновременно, а манипулировать списком транзакций из банковского счета может быть нелегко. Эти представления должны быть отделены.

Что касается максимальной глубины 4, это также вполне разумно, если это немного выше рекомендуемого уровня. Такая глубина является прямым результатом группировки с использованием подструктур, чтобы обеспечить читабельность данных. Как показано на рис. 7.14, детализация данных имеет два измерения: число свойств и глубину.

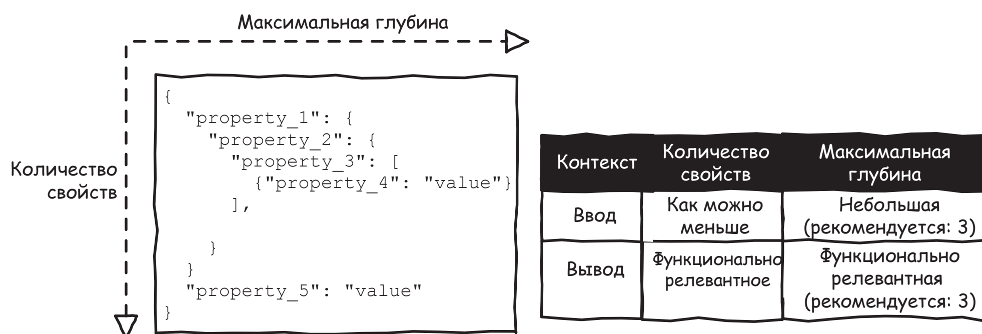



**Рис. 7.14.** Выбор числа свойств и максимальной глубины

Число свойств, которое API-интерфейс может возвращать в структуре данных – и это разумно, – является вопросом функциональной значимости; все предоставленные свойства должны быть функционально приемлемыми в контексте, в котором они используются. Однако, чем больше данных возвращает API, тем больше осторожности должен проявлять проектировщик в отношении организации (вспомните раздел 7.1.1) и актуальности, потому что, хотя все они уместны, наличие большого числа свойств не облегчает их использование.

По своему опыту я бы сказал, что, если свойств более 20, определенно стоит подумать об их организации и, возможно, бросить вызов каждому из них. Но это не универсальное средство; существуют области, где такое количество свойств, возможно, необходимо. Попробуйте определить собственные правила на основе своей области. Также напомню, что, как мы видели в разделе 5.2.1, необходимо запрашивать как можно меньше данных, чтобы обеспечить удобство использования. Количество свойств является весьма важным в этом контексте.

Что касается глубины, у нас здесь та же двойственность ввода/вывода; но в обоих случаях рекомендуется не выходить за пределы трех уровней глубины. Наличие более трех уровней усложняет манипуляцию необра-

ботанными данными, написание кода и чтение документации. Повторюсь, возможно, это правило нужно адаптировать к вашему контексту.

Организация данных также помогает упростить понимание вашего API, поэтому вам нужно будет найти баланс. Следите за детализацией данных, но помните, что в основном это вопрос функционального контекста, а не цифр. Однако детализация важна не только для данных, но и для целей.

### 7.2.2 Выбор детализации целей

Посмотрите еще раз на рис. 7.15. Стоит ли включать транзакции в представление банковского счета? Возможно, нет.

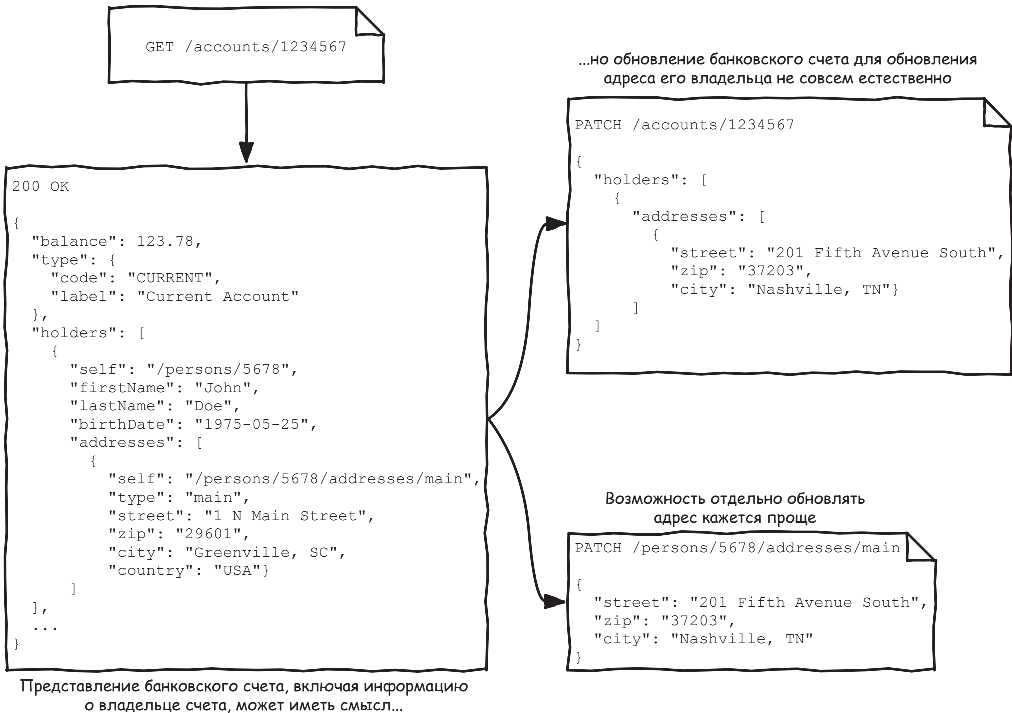



**Рис. 7.15.** Разная детализация целей при чтении и изменении данных

Если цель «Получить банковский счет» возвращает счет *и* список ее транзакций, нам придется иметь дело с управлением потенциально большим количеством транзакций. Всегда возвращать все транзакции может быть затруднительным, а управление разбиением транзакций на страницы может быть сложным (как вы узнали в главе 6). Запросы `GET /account/{id}?page=2` или `GET /account/{id}?transactionsPage=2` выглядят довольно неуклюже. Лучше предоставить отдельную цель для получения транзакций по банковскому счету (`GET /accounts/{id}/transactions`).

Выбор правильной детализации ваших целей заключается в обеспечении того, чтобы цель не делала две (или более) совершенно разные

вещи. Обратите внимание, что детализация целей не всегда согласована. Как показано на рис. 7.15, она может быть разной, например, при чтении или изменении данных.

Банковский API в настоящее время позволяет нам изменять адреса владельца счета путем обновления ресурса банковской счет. Хотя для этого ресурса может быть полезно предоставить информацию о владельце счета, включая его адреса, обновление адреса путем обновления ресурса на самом деле неестественно. Выполнение обновления таким образом означает сокрытие цели «Обновить адрес» в еще одной цели.

Здесь есть два проблемы. Во-первых, поначалу может быть неочевидным, что можно обновить адрес, обновив банковский счет. Во-вторых, данные владельца счета не зависят от конкретного банковского счета. Один и тот же владелец может владеть несколькими счетами, поэтому обновление адреса с помощью счета выглядит довольно неудобно. А что, если есть другие свойства банковского счета, которые можно обновлять через этот ресурс, такие как овердрафт?

Запрос минимальных входных данных и управление ошибками для цели «Обновить банковский счет» может стать довольно сложным как для проектировщиков, так и для потребителей, поскольку эта цель будет охватывать несколько подцелей. Было бы разумнее указать независимую цель для обновления адреса, как показано справа внизу на рис. 7.15, но вполне приемлемо предоставить доступ к сведениям об адресе через банковский счет, если это имеет смысл с функциональной точки зрения.

Помните, что детализация цели должна определяться контекстом и удобством использования.

Мы подробнее поговорим о детализации целей в главе 8, но вначале следует помнить, что, если детализация имеет значение для данных и целей, она, конечно же, важно и для API.

### 7.2.3 Выбор детализации API

Когда мы организовали цели банковского API в разделе 7.1.3, вокруг целей появились границы. Как показано на рис. 7.16, мы сгруппировали цели по категориям Account и Transfer.

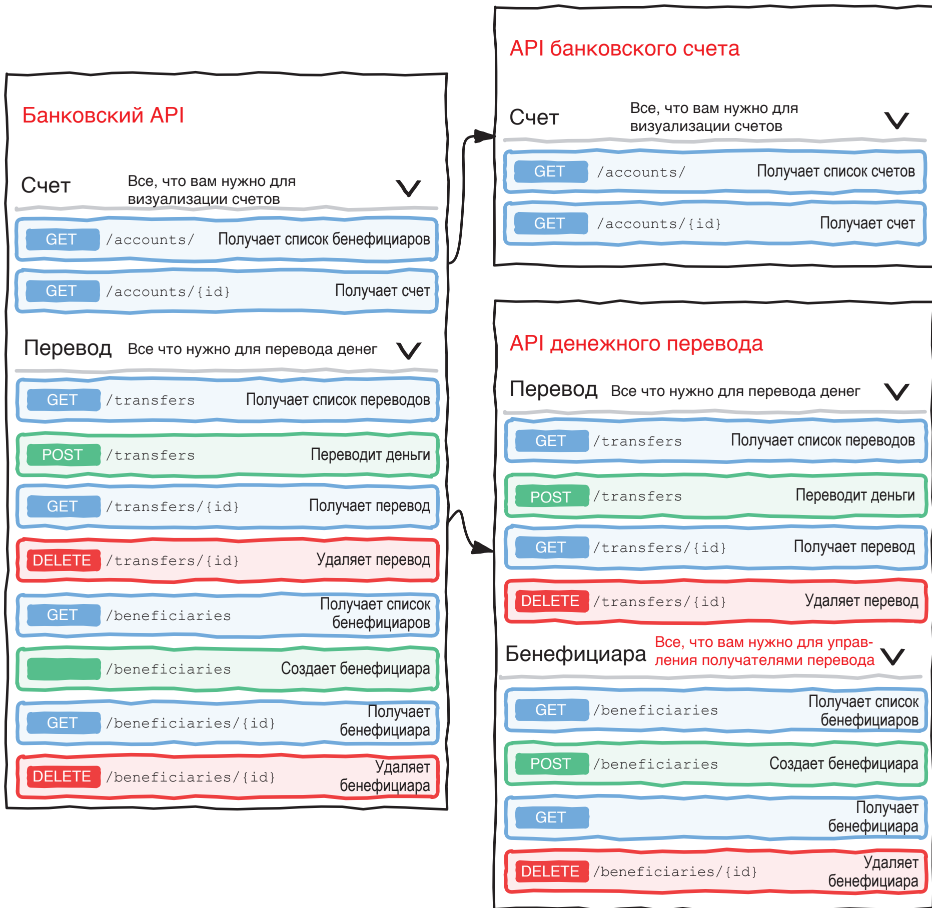



**Рис. 7.16.** От одного банковского API к отдельным API банковского счета и денежных переводов

Но это больше, чем просто категории. Каждая группа целей может быть полностью независимой. Поэтому, почему бы нам не разделить банковский API на два менее крупных, но функционально полезных API? Такие API будут проще в управлении, и их можно использовать независимо друг от друга в разных контекстах. Обратите внимание, что в API для денежных переводов цели были сгруппированы по начальным категориям `Transfer` и `Beneficiary`. Теперь имеет смысл организовать их в более мелкие группы.

Мы работали с представлением программного интерфейса, но организация и разделение целей API можно осуществлять на первых этапах проектирования при идентификации целей. Попробуйте применить полученные знания здесь к API онлайн-магазина, который мы спроектировали в главах 3 и 4. Как бы вы организовали и разбили список целей, показанный в табл. 7.2, на независимые менее крупные API? Вы должны заполнить столбцы Category и API; попробуйте придумать как минимум две разные версии. (Подсказка: если вы будете думать о том, кто пользователи, это может помочь вам найти одну из версий.)

**Таблица 7.2. Как бы вы организовали и разбили этот список целей**

| Цель | Категория | API |
|---|---|---|
| Создать пользователя | | |
| Поиск товаров | | |
| Получить информацию о товаре | | |
| Добавить товар в корзину | | |
| Удалить товар из корзины | | |
| Оформить заказ | | |
| Получить информацию о корзине | | |
| Список заказов | | |
| Добавить товар в каталог | | |
| Обновить товар | | |
| Заменить товар | | |
| Удалить товар | | |
| Получить статус заказа | | |
| Обновить пользователя | | |
| Удалить пользователя | | |

Помните, что, как только API будет организован в группы целей при идентификации целей или проектировании программного интерфейса, его можно разделить на более мелкие, но функционально важные API, которые можно использовать независимо.

На этом вторая часть этой книги подошла к концу; теперь вы знаете, как проектировать удобные для использования API. Это уже здорово, но не будем останавливаться на достигнутом: нам еще многое предстоит сделать. В третьей части этой книги вы научитесь проектировать API, принимая во внимание окружающий их контекст.

## *Резюме*

- Организуйте свойства данных, сортируя их, присваивая им имена с использованием шаблонов или группируя их в структуры данных.
- Распределите ответные сообщения по категориям и отсортируйте их по важности.
- Группируйте цели, ориентируясь на функциональность, а не на представления; вы можете использовать возможности формата описания API или шаблоны именования (OpenAPI-теги и URL-префиксы для REST API).
- Сохраняйте минимальное число свойств и уровней глубины в структурах данных.
- Избегайте создания целей типа *все-в-одном*.
- Разделяйте структуры данных, цели и даже API-интерфейсы на более мелкие, но функционально важные элементы, когда это возможно.

# Часть 3

# Контекстное проектирование API

Прочитав первые две части этой книги, вы научились проектировать API-интерфейсы, которые имеют смысл для потребителей, – которые легко понять и использовать, даже не задумываясь об этом. Это все, что вам нужно для проектирования API? Точно – нет. Было бы ужасной ошибкой полагать, что работа проектировщика API на этом закончена. Если остановиться здесь, это неизбежно приведет к созданию непригодных и даже опасных API. Ведь мы проектировали API, не принимая во внимание весть тот контекст, который их окружает.

- Является ли наш дизайн полностью безопасным с учетом предполагаемых функций API, и как он будет представлен пользователям?
- Будет ли он удобен в использовании, скажем, для мобильного приложения в сети 3G низкого качества?
- Действительно ли наш дизайн лучше всего подходит нашим целевым потребителям?
- Он реализуем на наших существующих системах?
- Наш дизайн настолько хорош, что не требует никакой документации?
- Совпадает ли наш дизайн со всеми другими нашими API?
- Что, если мы захотим обновить его после его запуска в производство – легко ли будет это сделать?

На все эти вопросы нужно ответить. Будучи проектировщиками API, мы должны учитывать весь контекст, окружающий API. Надеюсь, в последующих главах вы научитесь делать это.

Сначала поговорим о безопасности. Безопасность API – не запоздалая мысль, делегированная кому-то другому после проектирования. API должен быть исходно безопасным (*secure by design*), чтобы гарантировать, что

потребители, конечные пользователи и кто-либо или что-либо еще не могут делать или видеть больше, чем они должны. Это требует от проектировщиков API понимания механизмов безопасности API, умения разбивать API на части для облегчения контроля доступа, учитывать вопросы безопасности при осуществлении проектирования с точки зрения потребителей и, самое главное, знать, как обращаться с важными данными.

Тогда мы увидим, что API – это живая вещь; она будет бесповоротно эволюционировать. Проектирование модификаций API требует особой осторожности, чтобы избежать внесения критических изменений, которые вынудят всех потребителей обновлять свой код, чтобы иметь возможность использовать обновленный API. Знание того, как это сделать, является ключевым навыком для проектировщиков API, но умение проектировать API, которые способны развиваться с нуля, и снижение риска критических изменений при модификации еще более важно.

После этого мы поговорим об ограничениях сети. Мобильное приложение, работающее на смартфоне в сети 3G, не имеет тех же ограничений, что и приложение, работающее на сервере в локальной сети. Проектировщики API должны убедиться, что их он действительно пригоден для целевых потребителей в их среде, а также что дизайн действительно эффективен для всех вариантов использования, включая пограничные случаи, что может, например, включать в себя больше данных или больше вызовов.

Мы также увидим, что, если эффективность сети является основным ограничением, то лишь одним из многих. Проектирование API требует, чтобы мы знали обо всех ограничениях как со стороны потребителя, так и со стороны поставщика, чтобы создавать полностью удобные для применения и реально реализуемые API.

В последних двух главах вы узнаете, что проектировщикам API нужно больше, чем просто проектировать API. Мы поговорим о документировании. Независимо от того, насколько хорош дизайн, он должен быть задокументирован, чтобы не только потребителям, но и заинтересованным сторонам было проще понять его. API также должен быть задокументирован, чтобы люди, отвечающие за его реализацию, могли его точно создать. Как и безопасность, документация по API не должна быть запоздалой. Это нельзя полностью делегировать другому лицу после проектирования; проектировщики API должны участвовать в этом.

И наконец, мы увидим, как проектировщики API могут способствовать росту поверхности API-интерфейса организации. Рассматривая проектирование API на разных этапах с разных точек зрения, команда может гарантировать, что полученный API действительно будет таким, как ожидалось. Проектировщики API должны участвовать в этих обзорах, даже когда речь идет об API, над которыми они *не* работают. Также довольно часто многие проектировщики API работают над несколькими API в одной организации. Это требует, чтобы все они делились тем, что они делают и как они это делают – путем создания руководящих принципов проектирования и сообщества, – чтобы обеспечить определенную согласованность и не терять времени, заново изобретая колесо.

# 8 Проектирование безопасного API

**В этой главе мы рассмотрим:**

- пересечение безопасности и проектирования API;
- определение удобных для пользователя групп для управления доступом;
- адаптацию дизайна API для удовлетворения потребностей контроля доступа;
- адаптацию дизайна API для обработки конфиденциальных вещей.

Проектирование API-интерфейсов, которые имеют смысл для их пользователей и удобны в применении, безусловно, важна, но этого нельзя делать без учета безопасности. Безопасность API – не запоздалая идея, которой займутся позже (когда бы то ни было) специалисты по безопасности (кем бы они ни были). Проектирование и безопасность неразрывно связаны, когда речь идет о создании API или чего-либо еще.

Регулярно появляются новости о том, что какую-то компанию «взломали» через ее API, в особенности используя закрытые API для мобильных приложений. Я ставлю кавычки вокруг слова «взломали», потому что иногда такой взлом находится на уровне детского сада. В некоторых случаях хакеры просто проверяют ответы API и обнаруживают конфиденциальные данные, которые никогда не должны выходить из систем поставщика. А также этот классический вопрос: «Что произойдет, если я изменю идентификатор пользователя в запросе?»…«Я получаю данные других пользователей!»

Не потому, что API является *закрытым* или предназначен для партнеров и используется только доверенными потребителями, поэтому он может предоставить доступ к чему угодно, не задумываясь о безопасности. К безопасности открытого API обычно относятся более серьезно,

пока вовлеченные лица действительно знают, что она означает. Безопасность важна для всех типов API; и, будучи проектировщиком API, вы должны сыграть свою роль в этом.

По крайней мере, вы должны иметь базовое понимание безопасности API, чтобы проектировать безопасные API и эффективно общаться с людьми, участвующими в их создании. На эту тему можно было бы написать целую книгу, а может даже и несколько. Цель этой главы – дать общий обзор без подробного описания фактической реализации безопасности API. Такие книги, как «OAuth 2 в действии» Джастина Ричера и Антонио Сансо или «Безопасность API в действии» Нила Мэддена, опубликованные издательством Manning (https://www.manning.com/books/oauth-2-in-action и https://www.manning.com/books/api-security-in-action соответственно), дают более подробную информацию по этой теме. Хотя это обсуждение, очевидно, будет далеко не полным, его должно быть достаточно, чтобы помочь вам понять, как можно создавать исходно безопасные API. На рис. 8.1 подробно представлен API-вызов с точки зрения безопасности, чтобы проиллюстрировать, где сталкиваются безопасность API и проектирование.

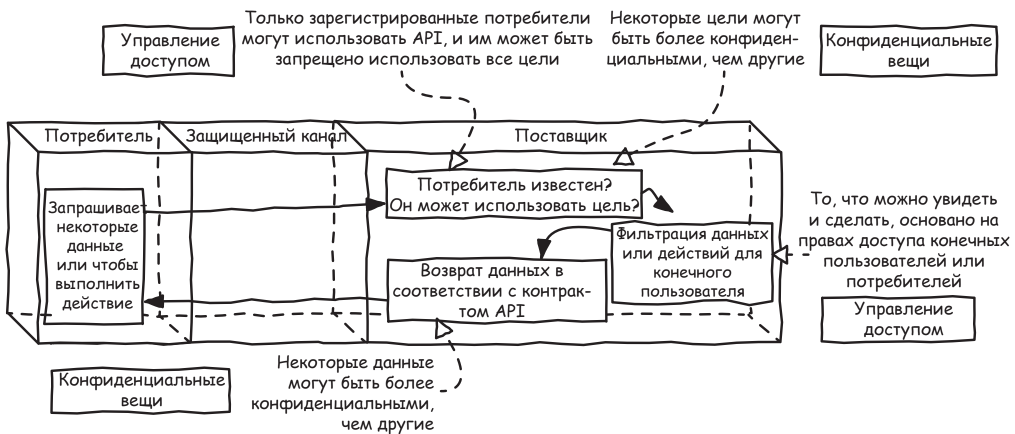



**Рис. 8.1.** API-вызов с точки зрения безопасности

Когда поставщик банковского API получает API-вызовы, он должен осуществлять контроль высокоуровневого доступа, чтобы быть уверенным в том, что потребители известны и им разрешено использовать запрошенную цель API. Затем поставщик выполняет ряд мер по контролю доступа более низкого уровня, чтобы гарантировать, например, что конечные пользователи видят только собственные счета.

В некоторых случаях запрос и ответ могут содержать конфиденциальные личные данные, с которыми следует обращаться осторожно. Сама по себе цель также может быть деликатной, как например, перевод денег, и требовать дополнительного внимания. *Контроль доступа* и *чувствительные данные* являются основными области, где безопасность и проектирование API пересекаются. Мы займемся этими вопросами, но сначала нам нужно подробнее поговорить о безопасности API.

## 8.1 Обзор безопасности API

Для проектирования безопасных API необходимо знать некоторые базовые принципы безопасности API. Важно также понимать, что означают эти принципы не только для разработчиков потребителей и поставщиков API, но и для конечных пользователей. Мы изучим все это, пройдя три этапа первого вызова API: регистрация потребителя, получение учетных данных, позволяющих использовать API, и выполнение вызова. Помимо экосистемы API, мы также познакомимся с фреймворком авторизации OAuth 2.0, определенным в RFC 6749 (https://tools.ietf.org/html/rfc6749). Хотя это и не единственный вариант, обычно он используется для защиты веб-API.

### 8.1.1 Регистрация потребителя

Безопасные API-интерфейсы позволяют использовать их только известным пользователям. Когда разработчики хотят использовать API в своих пользовательских приложениях, будь то мобильные или серверные приложения, они должны сначала зарегистрировать их. Для этого они могут использовать *портал разработки* API-поставщика.

Идеальный портал для разработчиков – это сайт, предлагающий документацию, учебные пособия, часто задаваемые вопросы, форумы, поддержку и любые другие полезные ресурсы и инструменты, которыми разработчики могут воспользоваться, чтобы понять, что делает API и как его использовать. Он также предоставляет информацию об использовании API; разработчики могут проверить, как их потребители используют API, а также получить доступ к журналам вызовов API. Разработчики должны зарегистрироваться самостоятельно (и, возможно, предоставить информацию о платеже, если API не является бесплатным). Как только это будет сделано, они смогут зарегистрировать своих потребителей. На рис. 8.2 показан процесс регистрации базового потребительского приложения для банковского API.

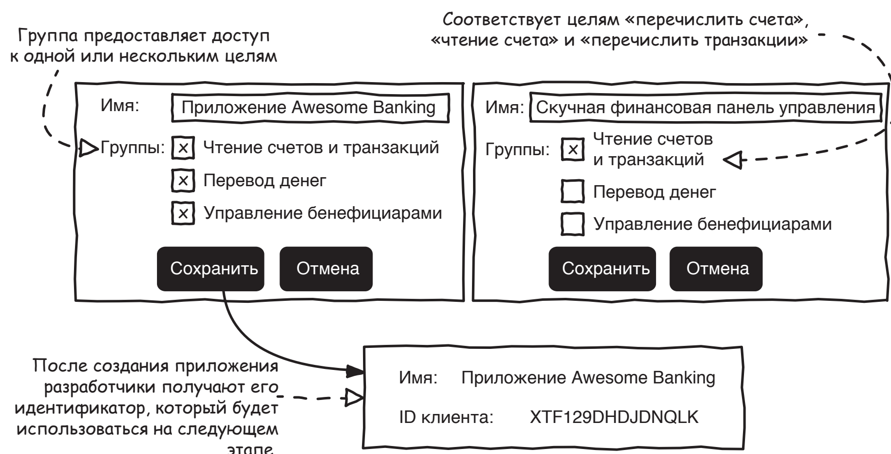



**Рис. 8.2. Регистрация потребителя, чтобы выбрать, какая часть API будет использоваться, и получить учетные данные**

Каждому потребителю дается имя. Здесь у нас есть «Приложение Awesome Banking» и «Скучная финансовая панель управления». Затем разработчики должны выбрать, какие области API будут использовать их потребители. *Область* соответствует одной или нескольким целям API. Скучная финансовая панель управления использует только область чтения счетов и транзакций, что соответствует целям «Перечислить счета», «Чтение счета» и «Перечислить транзакции». Следовательно, этому потребителю будет разрешено использовать только эти три цели. Приложение Awesome Banking использует все области, и поэтому ему будет разрешено использовать все цели API. Как только эта конфигурация будет завершена, разработчики смогут получить идентификатор клиента для своих приложений, который будет использоваться на следующем шаге.

Не у всех API, особенно закрытых, есть портал для разработчиков. Ту же настройку можно осуществить, например, сохранив (в защищенном режиме, конечно же) имя приложения, области и учетные данные в базе данных. Какими бы ни были средства, после регистрации потребителей разработчики могут перейти к следующему шагу: получить учетные данные, позволяющие потребителям использовать API. Давайте посмотрим, что это значит для приложения Awesome Banking.

### 8.1.2 Получение учетных данных для использования API

Если вы когда-либо использовали кнопку **Зарегистрироваться через Google/Twitter/GitHub/Facebook**, то, что вы увидите сейчас, должно выглядеть знакомо. Awesome Banking – стороннее приложение, созданное компанией Awesome. Его конечные пользователи являются клиентами банковской компания, предоставляющей банковский API. Приложение Awesome Banking и его конечные пользователи должны пройти проверку подлинности, чтобы получить *токен*, содержащий учетные данные, позволяющие приложению использовать банковский API.

На рис. 8.3 показаны роли, которые играют различные вовлеченные стороны. (Это упрощенное и частичное представление неявного потока OAuth 2.0, который, помимо прочего, является одним из способов получения учетных данных, позволяющих использовать API. Это лишь пример, а не рецепт. Меры безопасности должны быть выбраны экспертами по безопасности. Не используйте этот метод, если вы не знаете, что делаете.)

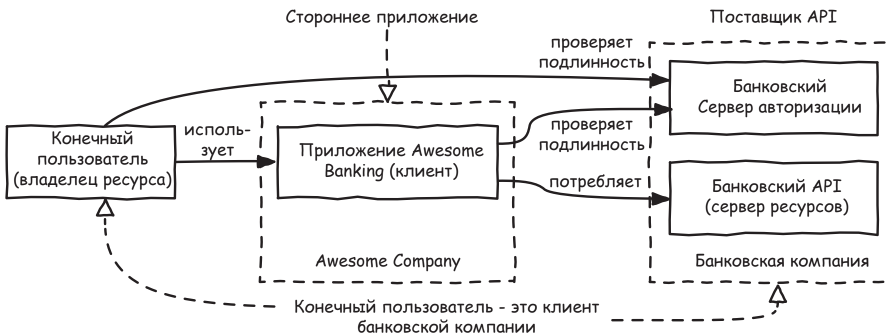



Рис. 8.3. Роли, участвующие в неявном потоке OAuth 2.0

*Сервер ресурса* – это приложение, предоставляющее банковский API. *Клиент* – это стороннее приложение Awesome Banking, которое использует этот API. *Владелец ресурса* – это конечный пользователь, использующий приложение Awesome Banking. И последнее, но не менее важное: *сервер авторизации* – это приложение, которое проверяет личность потребителя и конечного пользователя. На рис. 8.4 показано, как эти стороны работают сообща.

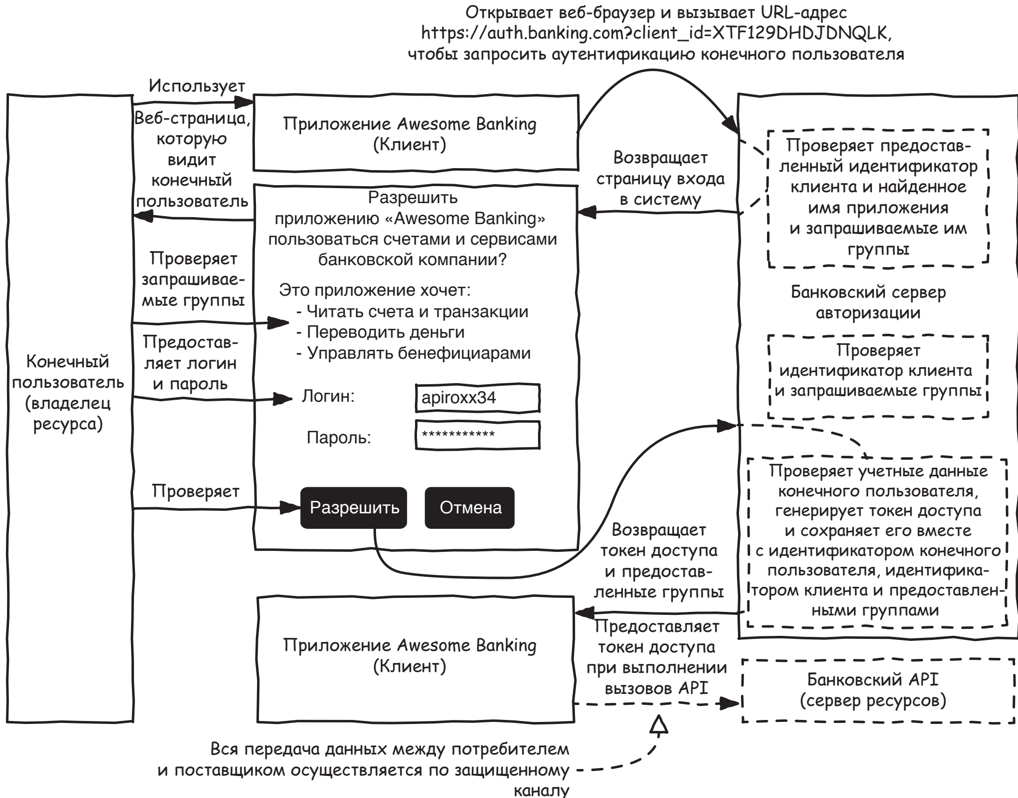



**Рис. 8.4.** Аутентификация приложения Awesome Banking и его конечных пользователей, в результате чего им предоставляется токен доступа, позволяющий использовать банковский API

Чтобы получить учетные данные, приложение Awesome Banking сначала связывается с сервером авторизации банковской компании для запроса аутентификации конечного пользователя. Обычно это делается путем открытия окна браузера с указанием URL-адреса сервера авторизации и включением идентификатора клиента в параметр запроса (например, https://auth.banking.com?client_id = XTF129DHDJDNQLK).

Когда сервер авторизации получает этот запрос, он проверяет, соответствует ли предоставленный идентификатор клиента известному потребителю. Если это так, он получает имя потребителя (приложение Awesome Banking) и запрашиваемые области действия (чтение счетов и транзакций, перевод денег и управление бенефициарами) в соответствии с конфигурацией, показанной на рис. 8.2. Затем он возвращает страницу входа

в систему, содержащую эту информацию. Конечные пользователи должны проверить, что то, что приложение хочет сделать (запрошенные области), – уместно. Если это так, они предоставляют свои логин и пароль, а затем нажимают кнопку **Авторизоваться**.

При нажатии на эту кнопку логин и пароль пользователя вместе со списком областей отправляются на сервер авторизации. Он проверяет, что запрашиваемые области соответствуют тем, что определены в конфигурации клиента (приложение Awesome Banking), и убеждается, что логин и пароль конечного пользователя действительны. Если все в порядке, генерируется *токен доступа*, учетные данные, которые позволяют приложению Awesome Banking использовать банковский API-интерфейс. Сервер авторизации хранит этот токен вместе с идентификатором конечного пользователя, идентификатором клиента и предоставленными областями; и наконец возвращает токен доступа приложению.

Обратите внимание, что любое взаимодействие между потребителем и поставщиком осуществляется по защищенному каналу, гарантируя, что никто не сможет перехватить обмен данными. При использовании протокола HTTP это делается с использованием протокола TLS (ранее известного как уровень защищенных сокетов или SSL). Если вы когда-либо использовали URL-адрес вида https://example.com, это значит, что вы использовали TLS.

### OAuth 2.0 и OpenID Connect

Согласно RFC 6749 (https://tools.ietf.org/html/rfc6749), «фреймворк для авторизации OAuth 2.0 позволяет стороннему приложению получать ограниченный доступ к пользовательским аккаунтам на HTTP-сервисах. Он работает по принципу делегирования аутентификации пользователя сервису, на котором находится аккаунт пользователя, позволяя стороннему приложению получать доступ к аккаунту пользователя».

Этот фреймворк предоставляет различные потоки для получения доступа к API от имени владельца ресурса. Каждый поток имеет свои плюсы и минусы и должен использоваться в правильном контексте. Это фреймворк только для авторизации; OAuth 2.0 не предоставляет никакой информации о том, как пользователи аутентифицируются (идентифицируются). OpenID Connect (https://openid.net/connect/) – протокол аутентификации на базе OAuth 2.0, который предоставляет такие возможности.

Теперь, когда у приложения Awesome Banking есть свой токен доступа, он может приступить к использованию банковского API. Об этом пойдет речь в следующем разделе.

## *8.1.3 Выполнение API-вызова*

В качестве первого вызова банковского API приложение Awesome Banking может запросить список счетов пользователя, как показано на рис. 8.5.

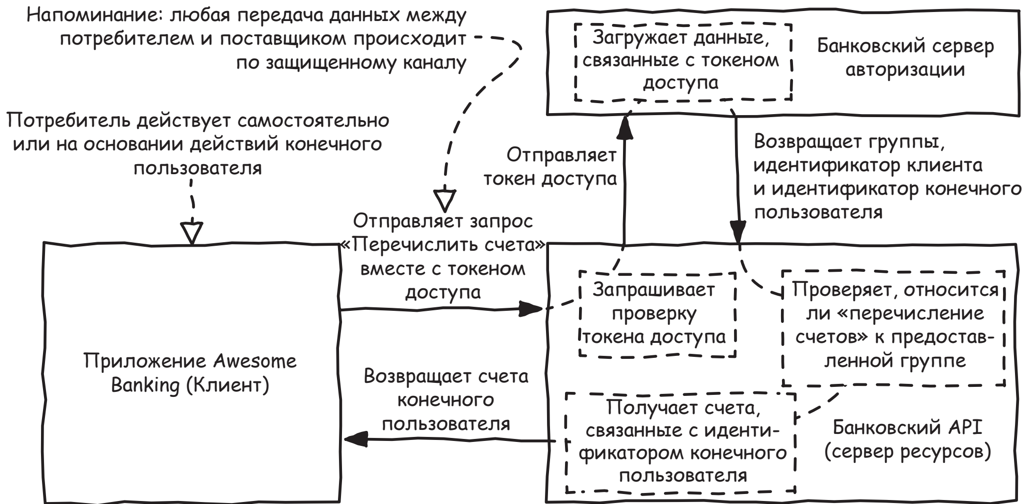



**Рис. 8.5.** Приложение Awesome Banking запрашивает список счетов

Для этого оно отправляет запрос перечислить счета банковскому API (конечно, по защищенному каналу). Этот запрос также содержит токен доступа, полученный ранее. Когда банковский API получает этот запрос, он связывается с сервером авторизации, чтобы проверить, действителен ли этот токен. Если это так, сервер авторизации может вернуть прикрепленные к нему данные, такие как идентификатор конечного пользователя, идентификатор клиента и предоставленные области.

Для начала проверяется, что цель «Перечислить счета» принадлежит одной из областей, предоставленных приложению Awesome Banking. Поскольку область чтения счетов и транзакций была предоставлена конечному пользователю, как описано в предыдущем разделе, реализация переходит к следующему этапу.

Потребитель запрашивает цель «Перечислить счета» без дальнейшего объяснения. Нужно ли возвращать все доступные счета? Конечно, нет! Запрос потребителя выполняется в контексте, определяемом данными, прикрепленными к токену доступа. Идентификатор конечного пользователя прикрепляется к токену доступа, и реализация использует его для фильтрации счетов. Таким образом, возвращаются только те счета, которые должны быть возвращены пользователю, выполняющему запрос.

Мы прошли все этапы нашего первого API-вызова и понимаем механизм, но давайте посмотрим, что все это значит, с точки зрения разработчика API.

### 8.1.4 Проектирование API с точки зрения безопасности

На рис. 8.6 собраны базовые принципы безопасности, которые вы встречали до сих пор и которые влияют на проектирование API.

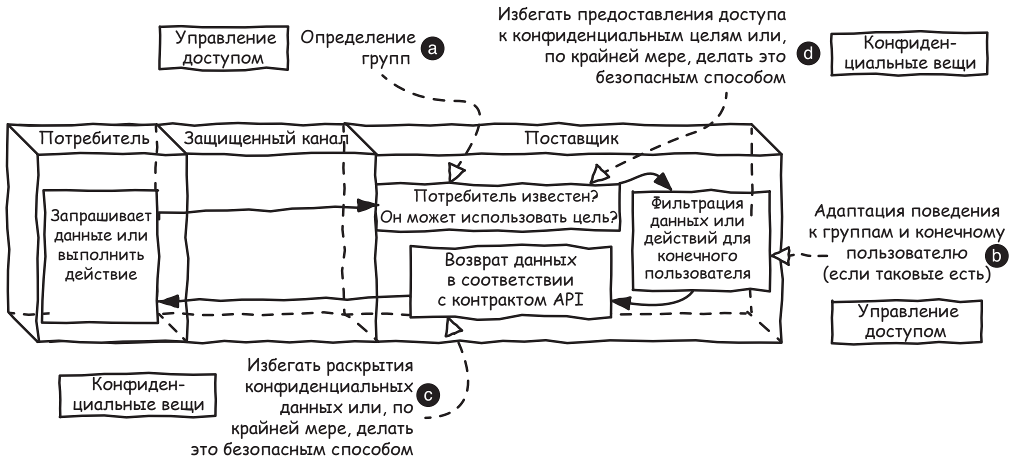



Рис. 8.6. Как пересекаются безопасность и проектирование API

Вы убедились, что весь обмен данными между потребителем и поставщиком должен происходить по защищенному каналу, гарантируя тем самым, что данные нельзя перехватить. Это не имеет ничего общего с проектированием API, но всегда полезно знать, что обмен данными безопасен. Вы также видели, что только зарегистрированным пользователям можно разрешить использовать API. Этим потребителям должно быть разрешено использовать только те части API, которые им действительно нужны и к которым конечные пользователи запрашивали доступ.

С другой стороны, реализация должна знать, какой потребитель запрашивает цель, чтобы убедиться, что ему действительно разрешено использовать эту цель. Определение областей (a), разбиение API для предоставления доступа только к выбранным целям нельзя осуществить без участия проектировщиков API, потому что эти группы целей должны иметь смысл для разработчиков, а также и для конечных пользователей. Помимо знания, разрешено ли потребителю использовать цель, реализация (b) также должна знать, от чьего имени делается запрос, чтобы адаптировать его поведение к правам конкретного конечного пользователя. Если конечные пользователи действительно вовлечены, это не всегда так. В этом заключается задача реализации, но проектировщики должны помнить об этом, потому что это может повлиять на проектирование API.

И это все? Проектировщикам API нужно думать только об управлении доступом для разработки безопасных API? Нет. Есть еще один, гораздо менее очевидный аспект безопасности API, который важен для проектировщиков.

Приложение Awesome Banking может перечислять счета через безопасное соединение только в том случае, если оно предоставляет действительный токен доступа, а области, прикрепленные к этому токену, охватывают эту цель. Возвращаемые счета – это только те счета, которые принадлежат конечному пользователю, подключенному к токену досту-

па. Звучит абсолютно безопасно, не так ли? Но что, если для каждого счета возвращенные данные содержат список всех прикрепленных к нему дебетовых карт? А что, если в случае с каждой картой предоставленные данные будут включать в себя ее номер, защитный код, дату истечения срока действия и полное имя владельца счета? Это было бы огромной проблемой, потому что это очень важные данные.

Должен ли банковский API предоставлять такие данные? Вероятно, нет. Рассматривать проектирование API с точки зрения безопасности – это не только вопрос контроля доступа; это также включает в себя вопрос: «Действительно ли мы должны предоставлять эти данные (c) через наш API?» И этот вопрос важен не только для данных, но и для целей (d).

Таким образом, для создания исходно безопасных API-интерфейсов проектировщики должны позаботиться об управлении доступом к приложениям, контроле доступа конечного пользователя и конфиденциальных данных. Давайте посмотрим, как можно это сделать.

## *8.2 Разделение API на части для облегчения управления доступом*

Давайте начнем с одного из самых очевидных аспектов безопасного проектирования API: как разделить API для облегчения управления доступом. Но сначала давайте поговорим о том, почему мы должны группировать цели по областям, чтобы предоставить потребителям доступ только к избранным целям, и почему нужно позаботиться о проектировании этих групп.

В реальном мире отель часто рассматривается как единое целое (особенно его гостями). Помимо номеров, отели могут предоставлять такие услуги, как бассейн, фитнес-центр, сауна или спа. Эти услуги обычно предлагаются всем гостям; они включены в стоимость номера. Но иногда некоторые из них являются необязательными. Гости должны сообщить персоналу отеля, что хотят ими пользоваться и заплатить за них в дополнение к стоимости номера. Некоторые из услуг могут даже быть доступными для людей, которые не проживают в отеле. Эти клиенты платят за пользование только этими услугами без использования остальных возможностей, предоставляемых отелем. Это означает, что, с точки зрения гостей отеля и не гостей, отель – это всего лишь бизнес по предоставлению услуг, которые могут быть доступны независимо друг от друга. На стойке регистрации отеля клиенты могут запросить доступ ко всем или некоторым из этих услуг. Тогда они смогут использовать только те услуги, к которым им предоставлен доступ.

Это может – и даже *должно* – происходить и в мире API. Как мы видели в разделе 8.1.1, два разных потребителя могут не использовать одни и те же цели. На рис. 8.7 показаны разные цели, используемые приложением Awesome Banking и скучной финансовой панелью управления.

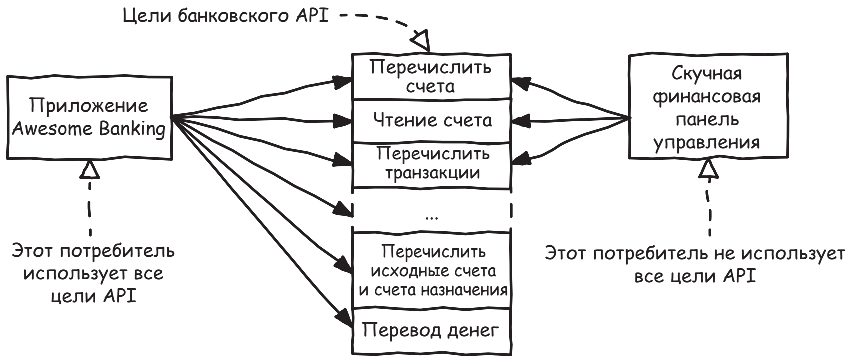



**Рис. 8.7. Разные потребители с разными потребностями не используют одни и те же цели**

Приложение Awesome Banking использует все доступные цели, в то время как скучная финансовая панель управления использует только цели «Перечислить счета», «Чтение счетов» и «Перечислить транзакции» без необходимости использовать другие цели. Но значит ли это, что этому потребителю нельзя позволять использовать эти другие цели? Почему мы реализуем такие средства управления доступом в API? Открытый проект обеспечения безопасности веб-приложений OWASP (https://www.owasp.org), всемирная некоммерческая благотворительная организация, занимающаяся повышением безопасности программного обеспечения, заявляет:

> *«Каждая функция, добавленная в приложение, добавляет определенный риск для всего приложения. Цель безопасного проектирования – снизить общий риск за счет уменьшения площади поверхности атаки».*
>
> *OWASP*

Ограничивая доступ потребителей только к тем целям, которые им действительно нужны, вы снижаете вероятность атаки. Это принцип *минимальных привилегий*, который также можно применять и к данным. Поскольку доступ к веб-API можно получить в интернете, чем меньше открытых дверей, тем лучше. Как и в случае с отелями, некоторые функции вашего API могут потребовать платной подписки, поэтому вам бы не хотелось видеть, как потребители, которые не внесли оплату, используют его бесплатно. Активация управления доступом облегчает процесс предоставления доступа к различным областям вашего API только избранным пользователям. Вот две веские причины для проектирования областей. Но мы также должны позаботиться о том, как мы их проектируем, чтобы обеспечить максимально возможный опыт взаимодействия для разработчиков и для конечных пользователей.

Когда разработчики регистрируют своих потребителей, что те могли использовать API, они должны выбирать соответствующие области. То же самое касается конечных пользователей: когда они позволяют сторонним приложениям получать доступ к API от их имени, они должны проверять запрошенные области и, возможно, выбирать их самостоятельно. Таким образом, цели API должны быть разделены на различные

группы, чтобы активировать механизмы управления доступом, и эти области должны быть тщательно спроектированы. Давайте посмотрим, как это можно сделать.

### *8.2.1 Определение гибких, но точных групп*

Во-первых, простой способ определения групп – это напрямую определить группу для каждой цели. При настройке своих потребителей через портал разработчиков банковского API разработчики могут видеть экраны конфигурации, как те, что показаны на рис. 8.8, позволяющие им выбирать группы (или цели в данном случае), которые им нужны для каждого потребителя.

При настройке приложения Awesome Banking, которое обеспечивает полный доступ ко всем службам банковского API, необходимо выбрать все группы. Для скучной финансовой панели управления, которая сосредоточена на анализе активности счетов, выбираются только группы по перечислению счетов и их чтению и перечислению транзакций. Настройка приложения PayFriend,

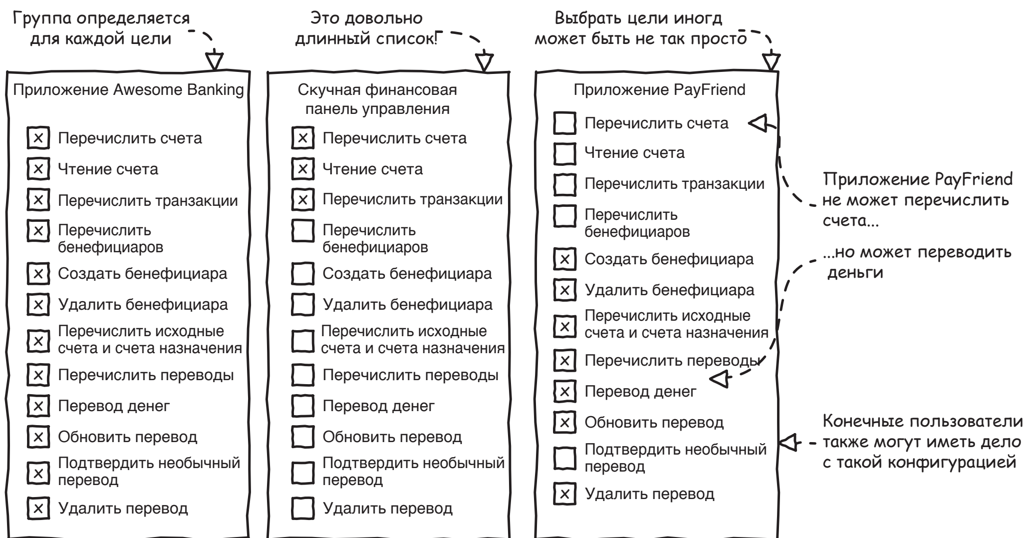



**Рис. 8.8. Управление доступом с помощью детализированных групп на базе целей**

которое предлагает своим пользователям с легкостью отправлять деньги друзьям, – хитрее. Цели, связанные с переводом денег, должны тщательно выбираться. Напротив пункта «Подтвердить необычный перевод» нет отметки, поскольку эта цель предназначена только для банковских консультантов, чтобы проверять определенные денежные переводы, считающиеся необычными, в зависимости от их назначения или суммы. После завершения настройки каждый из этих потребителей сможет использовать только те цели, которые соответствуют выбранным для них группам. Например, приложение PayFriend не может перечислить счета, но может инициировать перевод денег.

У нас есть 12 групп, соответствующие 12 целям банковского API. Конфигурация управления доступом довольно гибкая, но ее можно считать сложной. Каждую группу нужно тщательно выбирать. Что, если целей будет больше, а следовательно, и больше групп? Сложность увеличивается. А что, если позволить сторонним приложениям использовать API от имени своих конечных пользователей? Этим конечным пользователям также придется иметь дело с этой сложной конфигурацией. Процесс тщательного выбора нескольких групп из длинного списка может озадачить или вывести из себя как разработчиков, так и конечных пользователей.

Возможно, мы можем определить менее детализированные и более удобные для пользователя группы, основываясь на концепциях и действиях. Если вы проектировали свой API с использованием метода, описанного в разделе 3.2, это должно быть довольно просто, потому что вы уже делали что-то подобное. На рис. 8.9 показано, как определить такие группы для нашего банковского API.

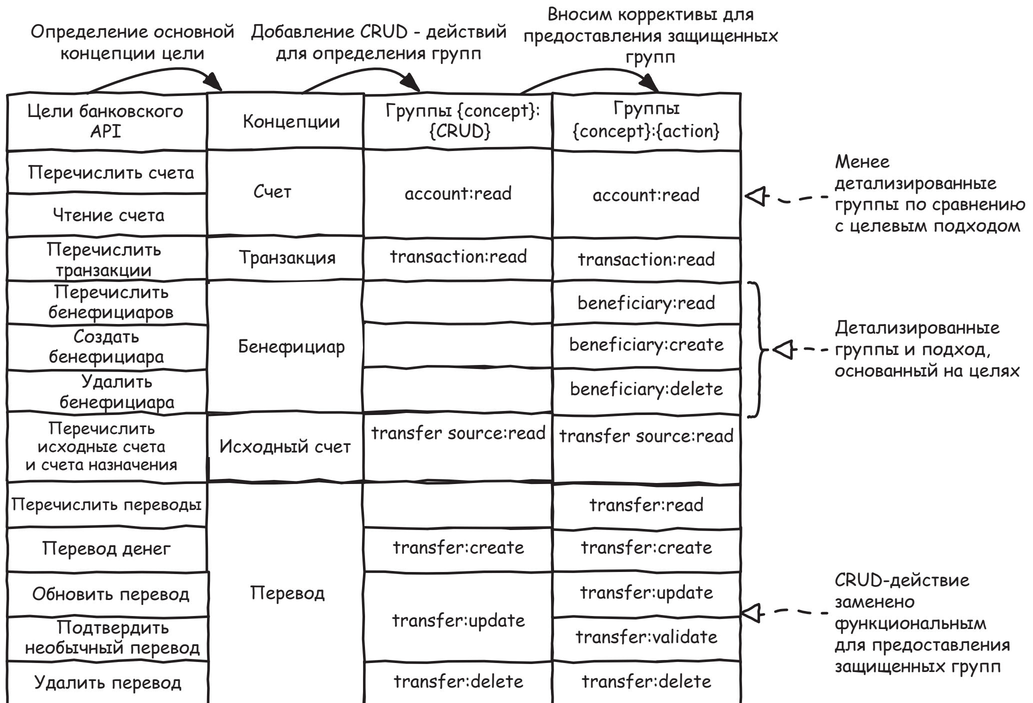



Рис. 8.9. Определение детализированных групп на основе концепций и действий

Первый шаг состоит в определении основной концепции (или ресурса) для каждой цели. Вы начинаете с определения основного существительного в цели. Например, цели «Перечислить счета» и «Прочитать счета» имеют дело с концепцией *счета*. Затем вы определяете действие акронима CRUD (Create, Read, Update, Delete), которое наилучшим образом представляет основной глагол цели. Для этих двух целей это – «чи-

тать»; следовательно, они попадают в группу `account:read`. Обратите внимание, что соглашение об именовании группы `{концепция}:{действие}` довольно распространено, но может быть не слишком удобным для пользователя. Такие имена обычно сопровождаются полезным описанием, таким как это:

```
"account:read": list accounts and access detailed information
about those
```

К сожалению, эта методика не всегда уменьшает число групп. Для целей, связанных с бенефициарами, у нас по-прежнему будет три группы, соответствующие целям «Перечислить бенефициаров», «Создать бенефициара» и «Удалить бенефициара». В некоторых случаях это может даже вызвать проблемы.

Цели «Обновить перевод» и «Подтвердить необычный перевод» обновляют денежный перевод и, следовательно, могут быть сгруппированы в рамках группы `transfer:update`. Но это было бы не очень безопасно! Позволяя потребителю обновлять денежный перевод, мы также позволяем ему использовать гораздо более важную цель – «Подтвердить необычный перевод». В этом случае было бы разумнее сохранить эту цель в рамках группы `transfer:validate`, которая использует настраиваемое действие вместо базовой функции CRUD.

Разделение на основе концепций и действий может привести к появлению групп, которые по-прежнему будут гибкими, но немного менее детализированными и сложными. Однако это нужно делать осторожно, чтобы избежать непреднамеренного предоставления неоправданного доступа к критически важным целям, и улучшение тут довольно минимальное. Давайте вспомним то, что мы узнали в главе 7 о проектировании краткого и хорошо организованного API. Можно ли использовать эти концепции, чтобы попытаться сгруппировать цели в более простые группы и обеспечить более удобное решение?

### 8.2.2 Определение простых, но менее детализированных групп

В разделе 7.1.3 вы узнали, как организовать цели по категориям. Почему бы не использовать их в качестве групп? На рис. 8.10 показано, как будут выглядеть эти группы на базе категорий для банковского API.

| Цели банковского API | Категории на базе группы |
|---|---|
| Перечислить счета | Счет |
| Чтение счета | |
| Перечислить транзакции | |
| Перечислить бенефициаров | Перевод |
| Создать бенефициара | |
| Удалить бенефициара | |
| Перечислить исходные счета и счета назначения | |
| Перечислить переводы | |
| Перевод денег | |
| Обновить перевод | |
| Подтвердить необычный перевод | |
| Удалить перевод | |

Возможно приемлемая группа

Гораздо менее детализированная группа, предоставляющая ненужный доступ

**Рис. 8.10. Определение групп на базе категорий – обычно плохая идея**

Группа «Счет» кажется приемлемой, но группа «Перевод» слишком обширная; она может предоставить неоправданный доступ к целям, связанным с бенефициарами, и критически важной цели «Подтвердить необычный перевод».

Следовательно, обычно категории не подходят для использования их в качестве основы для групп, поскольку имеют разные цели:

- *категории* – организовывают цели с функциональной точки зрения и помогите потребителям понять, как использовать API;
- *группы* – убеждаются, что потребители получают доступ только к тем целям, которые им действительно необходимы.

Если категории не сработали, как нам организовать наши цели в значимые, но безопасные группы? Можно попытаться строить их на том, чего пользователи могут достичь с помощью API. Такие группы более или менее соответствуют тому, что мы определили при заполнении таблицы целей API (см. раздел 2.3). На рис. 8.11 показана частичная таблица с целями API для банковского API с акцентом на переводы.

Некоторые столбцы были удалены из таблицы API-целей, чтобы было проще

| Участники | Действия | Цели | Разделение безопасности на основе ролей и функциональности |
|---|---|---|---|
| Владельцы счетов | Доступ к информации о счете | Перечислить счета<br>Чтение счета<br>Перечислить транзакции | Доступ к информации о счетах |
| | Перевод денег | Перечислить исходные счета и счета назначения<br>Перевод денег | Перевод денег |
| | Управление переводами | Перечислить переводы<br>Обновить перевод<br>Удалить перевод | |
| Банковские консультанты | Подтверждение необычных денежных переводов | Перечислить переводы<br>Подтвердить необычный перевод | Подтверждение переводов |

Группа, соответствующая действию

Группа, состоящая из двух действий

Группа, соответствующая действию

Рис. 8.11. Определение групп на базе ролей и функциональности

Эта таблица говорит нам, что для доступа к информации о счете владельцы счетов перечисляют счета, читают счет и перечисляют транзакции. Таким образом, можно было бы создать группу доступа к информации о счетах, соответствующую этому действию и включающую в себя эти три цели.

То же самое касается банковских консультантов, которые отвечают за проверку необычных переводов. Соответствующие цели могут быть сгруппированы в области проверки переводов. Таким же образом можно было бы поступить в случае перевода денег и управления переводами, но разделять их, возможно, и не имеет особого смысла. Например, после создания регулярных переводов у потребителей может возникнуть желание удалить их, когда они им больше не нужны. Поэтому, возможно, нужно создать группу «Перевод денег», включающую в себя две этих цели.

Чтобы попрактиковаться, я дам вам возможность дополнить таблицу целей API другими целями и соответствующим образом определить группы. Как видите, таким образом мы получаем меньше менее гибких, но более удобных для пользователя групп. Такие менее детализированные группы можно рассматривать как своего рода способ, позволяющий потребителям получить доступ к нескольким целям одновременно.

**ПРИМЕЧАНИЕ.** Такая стратегия определения групп на базе ролей и функциональности должна помочь вам избежать непреднамеренного предоставления доступа к критически важными целям, но на всякий случай всегда нужно все проверить дважды.

Вы также можете создавать совершенно *произвольные* группы. В некоторых случаях это может иметь смысл. Например, можно определить группы на уровне администратора для каждого ресурса: группа `beneficiary:admin`, предоставляющая доступ ко всем целям, связанным с бенефициарами, и т. д. При определении таких групп следует проявлять осторожность, потому что они могут разрешить неправомерный доступ, если он предоставляется приложениям без надлежащего внимания.

Вы увидели несколько разных способов определения групп. Какой из них нужно использовать?

## 8.2.3 Выбор стратегии

Какая стратегия является наиболее подходящей, когда речь идет об определении групп для активации управления доступом в API? Стратегия, при которой создаются гибкие, но сложные детализированные группы или менее гибкие, но более удобные для пользователя группы? И какой подход следует принять в рамках каждой стратегии? К сожалению, правильного ответа нет. В зависимости от потребителей, разработчиков и конечных пользователей API, наиболее подходящий подход может быть разным. Но есть и хорошие новости: возможно, вам и не нужно ничего выбирать!

Помимо создания менее детализированных и удобных для пользователя групп, стратегия, построенная на основе вопросов «что?», демонстрирует кое-что интересное. На рис. 8.12 видны цели, охватываемые двумя из только что определенных нами групп.

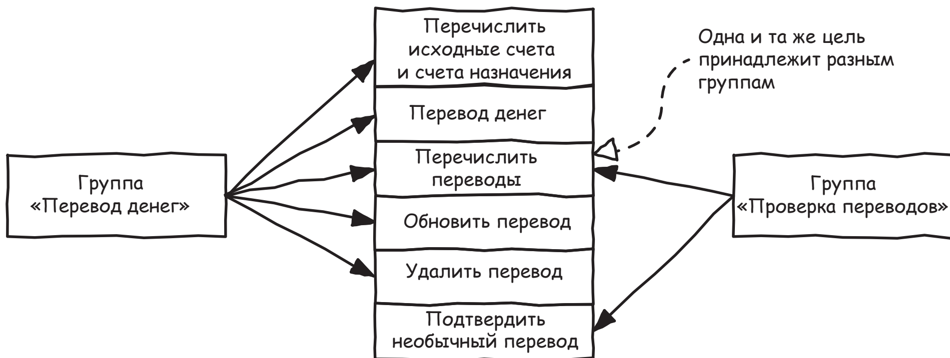



Рис. 8.12. Одни и те же цели могут принадлежать разным группам

Цель «Перечислить переводы» принадлежит двум группам! Это хорошая новость: она означает, что между группами может быть некое совпадение, что позволяет нам определять различные *уровни* групп.

Можно бы, например, использовать концептуальный и основанный на действиях подход, когда у нас есть такие группы, как `beneficiary:read` и `beneficiary:delete` наряду с произвольной группой, где все цели, охватываемые этими группами, также будут охвачены группой `beneficiary:admin`. Мы даже могли бы добавить более удобный для пользователя тре-

тий уровень, используя упомянутую выше стратегию для сторонних интеграций с участием конечных пользователей. Мы не обязаны показывать все свои группы API всем своим потребителям или допускать их до них; при необходимости можем предоставить разные представления доступных групп и, как мы уже делали раньше, использовать формат описания API для определения этих групп.

### *8.2.4 Определение групп с помощью формата описания API*

Если проектируемый вами API связан с форматом описания API, обязательно проверьте, позволяет ли он описывать группы. Как показано в приведенном ниже листинге, спецификация OpenAPI 3.0 позволяет это.

**Листинг 8.1 Описание групп**

```
components:
  securitySchemes:
    BankingAPIScopes: ①
      type: oauth2
      flows:
        implicit:
          authorizationUrl: ②
            "https://auth.bankingcompany.com/authorize"
          scopes: ③
            "beneficiary:create": Create beneficiaries
            "beneficiary:read": List beneficiaries
            "beneficiary:delete": Delete beneficiaries
            "beneficiary:manage": Create, list, and
              delete beneficiaries
```

① **Группы многократного использования определяются в разделе component.securitySchemes**

② **При необходимости можно указать фиктивный URL-адрес на этапе проектирования. (примечание: URL-адрес должен находиться на одной строке).**

③ **Схема безопасности может содержать несколько групп.**

Определение группы выполняется в разделе `component.securitySchemes` документа OpenAPI. Схема `BankingAPIScopes` определяет все потоки и группы, которые могут использоваться в банковском API при использовании типа безопасности `oauth2`. На данный момент доступен только поток `implicit`, и, когда он используется, разрешены только группы, связанные с бенефициарами.

В этом листинге определены четыре группы. Каждое определение группы сопровождается описанием. Первые три – это группы на базе целей: `beneficiary:create`, `beneficiary:read` и `beneficiary:delete`. Четвертая, `beneficiary:manage`, – произвольная группа, охватывающая три цели, связанные с бенефициарами. Хотя недостаточно просто определить эти группы. Они должны быть связаны с целями, как показано в приведенном ниже листинге.

**Листинг 8.2. Связывание цели с группами**

```
paths:
  /beneficiaries:
    get:
      tags:
        - Transfer
      description: Gets beneficiaries list
      security: ①
        - BankingAPIScopes: ②
        - "beneficiary:read"
        - "beneficiary:manage"
      responses:
        "200":
          description: The beneficiaries list
```

① **Перечисляем используемые схемы безопасности.**

② **Ссылаемся на схему безопасности из раздела component.securityScheme и перечисляем группы, необходимые для использования этой цели.**

Цель «Перечислить бенефициаров» представлена `GET /beneficiaries`. Она входит в группы `beneficiary:read` и `beneficiary:manage` схемы безопасности `BankingAPIScopes`, которую мы определили в листинге 8.1. Это означает, что эту цель могут использовать только потребители, которым был предоставлен доступ хотя бы к одной из этих двух групп.

Если API использует два типа групп, может быть полезно провести различие между ними в описании API. Это можно сделать просто для предоставления более четкой документации или для отображения только определенных типов групп, в зависимости от того, кто использует API, например. В приведенном ниже листинге показано, как сгруппировать их.

**Листинг 8.3. Объединение групп**

```
components:
  securitySchemes:
    ConceptActionBasedSecurity: ①
      type: oauth2
      flows:
        implicit:
          authorizationUrl: "https://auth.
          ➥ bankingcompany.com/authorize"
          scopes:
            "beneficiary:create": Create beneficiaries
            "beneficiary:read": List beneficiaries
            "beneficiary:delete": Delete beneficiaries
  ArbitraryBasedSecurity: ②
    type: oauth2
    flows:
```

```
        implicit:
          authorizationUrl: "https://auth.bankingcompany.
          ➥ com/authorize"
          scopes:
            "beneficiary:manage": Create, list, and
            ➥ delete beneficiaries
```

① **Содержит только целевые группы.**
② **Содержит группу более высокого уровня.**

И снова мы применяем то, о чем узнали в главе 7: при проектировании лаконичного и хорошо организованного API мы организуем компоненты API, чтобы облегчить понимание и использование. Схема безопасности `BankingAPIScopes` была разделена на две новые схемы, каждая из которых содержит определенный тип группы. В приведенном ниже листинге показано, как их использовать

**Листинг 8.4. Связывание цели с группами из разных объединений**

```
paths:
  /beneficiaries:
    get:
      tags:
        - Transfer
      description: Gets beneficiaries list
      security: ①
        - ConceptActionBasedSecurity:
          - "beneficiary:read"
        - ArbitraryBasedSecurity:
          - "beneficiary:manage"
      responses:
        "200":
          description: The beneficiaries list
```

① **Раздел security может содержать ссылки на различные схемы безопасности.**

Раздел `security` теперь содержит ссылки на схемы безопасности `ConceptActionBasedSecurity` и `ArbitraryBasedSecurity`, в каждой из которых перечислена используемая группа, как в листинге 8.2.

Чтобы попрактиковаться, вы можете расширить этот пример, добавив другие цели банковского API и три уровня групп (группа на основе целей, произвольная группа и группа на основе действий). Вы также можете добавить еще один тип потока OAuth, например учетные данные клиента (подсказка: см. описание «OAuth Flows Object» в документации к спецификации OpenAPI по адресу https://github.com/OAI/OpenAPI-Specification).

Теперь, когда вы увидели, как определять группы, давайте посмотрим, как на контракт интерфейса API могут повлиять вопросы управления доступом более низкого уровня.

## 8.3 Проектирование с учетом управления доступом

В наши дни постояльцам отеля обычно дают карты доступа вместо старых добрых ключей. Они могут использовать эти карты в лифтах, чтобы попасть на этажи, где расположены их номера, и, конечно же, для того чтобы открывать двери своих номеров. И очевидно, что они не могут использовать эти карты, чтобы открывать двери номеров других постояльцев.

У сотрудников отеля также есть карты, которые помогают попасть на зарезервированные для персонала этажи или в номера постояльцев. Разные сотрудники могут иметь доступ ко всем номерам отеля или только к тем, которые расположены на данном этаже. И постояльцы, и сотрудники могут попасть на этажи и в номера отеля, но уровень доступа у всех у них разный.

В мире API также существует низкоуровневое, детализированное управление доступом. В основном он обрабатывается реализацией, но проектировщики API также должны сыграть здесь свою роль. Чтобы гарантировать, что все идет гладко, проектировщики должны знать, какие данные необходимы для фактической реализации управления доступом, и адаптировать свои проекты, если это необходимо.

### 8.3.1 Какие данные необходимы для управления доступом

То, что Приложению Awesome Banking разрешено перечислять счета, не означает, что ему должно быть разрешено перечислять их *все*. Оно может извлекать только те счета, которые принадлежат его конечным пользователям, клиентам банковской компании. А что, если эта компания создаст приложение Bank Advisors, используя банковский API для своих банковских консультантов, управляющих счетами клиентов? Что должна вернуть цель «Перечислить счета» этому потребителю? Должна ли она вернуть все счета клиентов или только те, что относятся к клиентам, которыми управляют консультанты? Каким бы ни был ответ, цель «Перечислить счета» не будет вести себя одинаково при запуске для клиента или консультанта. Как показано на рис. 8.13, каждый потребитель имеет свое представление о банковских счетах, доступных с помощью цели «Перечислить счета».

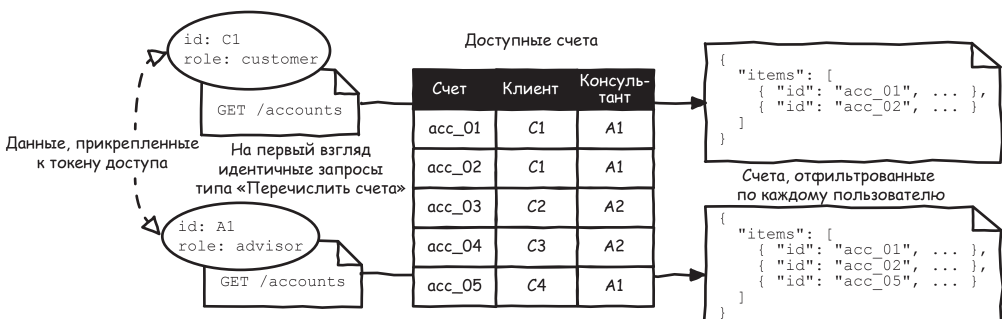



Рис. 8.13. Два, казалось бы, одинаковых запроса, дающих разные результаты

В этом примере дизайн API, похоже, не затронут. В обоих случаях представление цели «Перечислить счета» одинаково: в качестве REST API это будет что-то вроде `GET /accounts` – без каких-либо параметров. Его поведение будет изменено в соответствии с личностью конечного пользователя; но в запросе `GET /accounts` нет параметра `endUser`, так как мы узнаем, кто этот конечный пользователь? На самом деле этот параметр существует, но он скрыт.

Как вы видели в разделе 8.1.3, потребители должны отправить токен доступа со своим запросом, чтобы осуществить вызов API, так же как постояльцы отеля должны использовать свою карту доступа, чтобы открыть дверь. После своего создания токен хранится где-то вместе с некими данными. Эти данные являются скрытой частью контракта интерфейса API. Будучи проектировщиком API, вы должны знать об этом, чтобы быть уверенным, что то, что вы проектируете, действительно работает и безопасно.

Такие данные являются относительно стандартными; обычно они состоят из идентификатора потребителя, предоставленных групп, идентификатора конечного пользователя и, возможно, роли конечного пользователя (типа пользователя) или прав (что разрешено делать этому конкретному пользователю). Когда реализация получает запрос вместе с токеном, она может извлекать эти данные и осуществлять все необходимые функции управления доступом напрямую, основываясь на данных запроса или на основе каких-то других извлеченных данных.

Например, если приложение Awesome Banking отправляет запрос `GET /account/1234567`, реализация должна проверить, что конечный пользователь, чей идентификатор прикреплен к токену доступа, отправленному вместе с запросом, упоминается как один из владельцев счета 1234567. Однако, чтобы быть уверенным в этом, вам нужно будет проконсультироваться с ответственными за реализацию или, точнее, с ответственным за уровень безопасности. В нашем случае это довольно просто, но учтите, что механизм управления доступом не всегда такой прозрачный. Кроме того, цели могут потребоваться несколько разные представления для разных конечных пользователей или ролей.

### *8.3.2 Адаптация дизайна при необходимости*

В разделе 5.2 мы сосредоточились на цели «Перевести деньги», которая переводит некую сумму денег с исходного счета на целевой. Денежный перевод всегда инициируется по запросу клиента. Исходный счет принадлежит этому клиенту. Целевой счет – это счет, также принадлежащий этому клиенту или получателю, который был предварительно зарегистрирован клиентом. REST-представление этой цели состоит из запроса `POST /transfers`, тело которого содержит три свойства: `amount`, `source` и `destination`.

Если эта цель используется приложением Awesome Banking, которое используют клиенты банковской компании, идентификатор клиента прикрепляется к токену, отправленному вместе с запросом. Поэтому ре-

ализация может легко проверить, что все в порядке. Она проверяет, что исходный счет принадлежит клиенту, а также что целевой счет принадлежит тому же клиенту или бенефициару, зарегистрированному этим клиентом.

Но что, если эта цель «Перевести деньги» используется приложением Bank Advisor? Это приложение не используется клиентами, а только банковскими консультантами. Однако клиенты могут позвонить своим консультантам, чтобы сделать запрос на перевод денег. В этом случае идентификатор пользователя, прикрепленный к токену, является идентификатором консультанта, но не клиента.

Элементы управления в реализации менее просты, но все же осуществимы. Реализация проверяет, что исходный счет принадлежит клиенту, управляемому консультантом. Счет назначения должен принадлежать клиенту, который владеет счетом, или бенефициару, зарегистрированному тем же клиентом. Кажется, все в порядке, но есть одна проблема. Система должна отслеживать, какой клиент сделал запрос на перевод денег.

Если банковский счет принадлежит одному клиенту, идентификатор клиента легко найти, поскольку он привязан к банковскому счету, и только его владелец может сделать запрос на перевод. Но если счет совместный и принадлежит нескольким клиентам, это не сработает. Нужно изменить цель, чтобы передать реализации правильный идентификатор клиента.

Мы можем добавить необязательное свойство `customerId` в тело запроса `POST /transfers`, а также создать новое представление ресурса `transfer` и использовать для него метод `POST` следующим образом:

```
POST /customers/{customerId}/transfers
```

Второй вариант, кажется, имеет больше смысла, если отступить и посмотреть на API с более высокого уровня. Банковскому консультанту может потребоваться перечислить переводы клиента, поэтому метод `GET`

```
GET /customers/{customerId}/transfers
```

при использовании того же пути к ресурсам может быть полезным. Каким бы ни было выбранное решение, реализация теперь сможет проверить, что, когда цель «Перевести деньги» исходит от консультантов, это может осуществляться только для клиентов, для которых у них есть права делать это.

На самом деле, если вы используете таблицу целей API и метод, описанный в разделе 2.3, который должен был идентифицировать всех пользователей, что они делают, как они это делают и особенно то, что им нужно для этого, вы должны иметь возможность легко и почти незаметно справиться с дизайном API, по сравнению с вопросами, касающимися безопасности. Напомним, что этот метод снова показан на рис. 8.14.

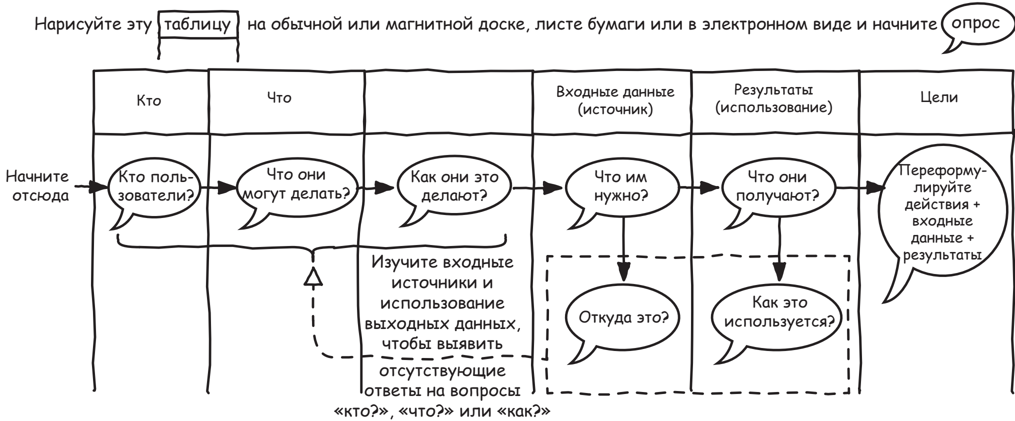



Рис. 8.14. Таблица целей API

Работа с группами показала, что, хотя проектировщики API должны избегать точки зрения поставщика в своих проектах, им иногда необходимо знать, что все-таки происходит в реализации.

Есть еще одна тема, касающаяся безопасности API, над которой нужно поработать, и, пожалуй, это самая важная тема с точки зрения проектировщика API. API предоставляет доступ к данным или возможностям. Поэтому проектировщик API должен проверить, являются ли они конфиденциальными, действительно ли нужно предоставлять к ним доступ, и, если это так, выбрать наиболее безопасный способ включить их в API.

## 8.4 Обработка конфиденциальных данных и важных вещей

Когда люди останавливаются в отеле, то обычно не берут с собой все свои вещи. Не имеет смысла приносить ценные или важные предметы, такие как декларация о подоходном налоге, драгоценности прапрабабушки или дорогой и редкий комплект маракасов для компьютерной игры Samba de Amigo для консоли Sega Dreamcast. Они вряд ли понадобятся и могут потеряться, либо их могут украсть. Но есть и другие ценные вещи (например, удостоверения личности, паспорта, телефоны или камеры), которые все нужно принести. Если эти вещи нужно оставить в номере отеля, их можно держать в сейфе номера. Наличные и кредитные карты также можно обезопасить подобным образом.

В мире API, как и в мире программного обеспечения в целом, всегда нужно проверять, касается ли то, что может быть запрошено, предоставлено или сделано с помощью API, конфиденциальным. Если это так, мы должны убедиться, что это действительно необходимо, а затем должны создать максимально безопасный дизайн. Это можно сделать либо путем настройки представления конфиденциальных данных, либо путем выбора адаптированных механизмов управления доступом. В первую очередь этот вопрос касается данных, целей и ответных сообщений, но мы также должны позаботиться и о базовом протоколе и архитектуре, используемой API.

## 8.4.1 Обработка конфиденциальных данных

Когда потребители используют цель «Прочитать счет», они получают подробную информацию о нем. Эта информация может состоять из номера счета, баланса и списка связанных с ним дебетовых карт. В системах банковской компании основная информация, связанная с картой, состоит из имени ее владельца, ее номера (который также носит название, первичный идентификатор счета, или PAN), даты истечения срока ее действия и CVV (кода проверки подлинности карты, расположенного на обратной ее стороне). Карты, предоставленные банковской компанией, могут быть заблокированы (это полезно, если вы считаете, что ваша карта может быть потеряна или украдена). И если клиенты хотят, они также могут определить месячный предел (они получают оповещение по SMS, электронной почте или уведомление, когда их общие ежемесячные платежи превышают этот уровень). Возвращаемые данные могут выглядеть так, как показано слева на рис. 8.15. Но даже если эти данные отправляются по защищенному соединению (см. раздел 8.1.3), действительно ли разумно возвращать все доступные данные?

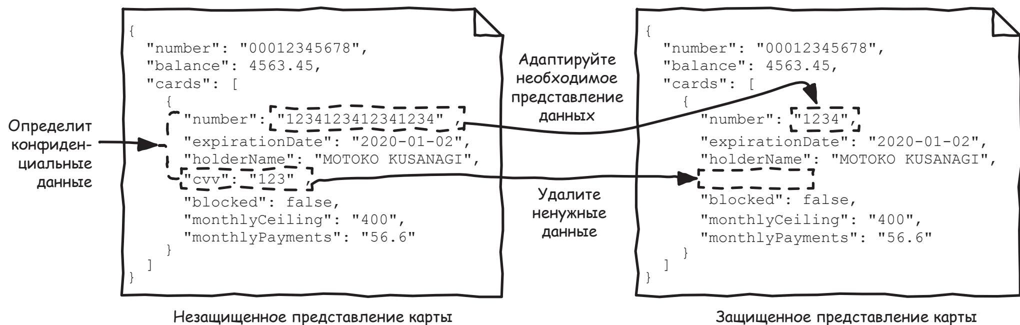



**Рис. 8.15. Проектирование безопасного представления банковской карты**

Месячный предел, ежемесячные платежи на текущий момент и флаг «заблокировано» – это не конфиденциальные данные; с ними никакого вреда причинить нельзя. С другой стороны, номер карты, CVV, имя держателя карты и срок действия являются весьма важными; эту информацию можно использовать для осуществления платежей в интернете или по телефону. Возможно, нужно подумать дважды, прежде чем возвращать всю эту информацию.

CVV используется только для онлайн-платежей, и он написан на самой карте, поэтому его можно удалить. Номер карты полезен для идентификации карты, но можно оставить только последние четыре цифры; этого достаточно для идентификации карты в контексте банковского счета. Можно оставить дату истечения срока действия и имя владельца карты, потому что это нужная информация. Теперь, когда мы удалили CVV и выбрали защищенное представление номера карты, никто не может воспользоваться этими данными, чтобы причинить вред. Как видите, идентифицируя элементы конфиденциальных данных и решая удалить их, адаптируя представление других, можно спроектиро-

вать безопасное представление дебетовой карты, которое остается значимым и актуальным.

Таким образом, первым этапом проектирования защищенного API-интерфейса является определение конфиденциальных данных, которые будут запрашиваться и предоставляться через этот API. Проблема состоит в том, что термин *конфиденциальные данные* охватывает широкий диапазон данных и может отличаться в зависимости от вашей области или отрасли. Идентифицировать конфиденциальные данные иногда довольно просто, как в примере с дебетовой картой, так как то, что следует считать конфиденциальным, довольно очевидно. Точно так же никто не будет предоставлять доступ к конфиденциальным данным, таким как пользовательские имена и пароли клиентов. Но иногда не все так очевидно.

Существует множество национальных, предметно-ориентированных и международных регламентов, стандартов или передовых практик, которые влияют на то, можете ли вы манипулировать данными и каким образом. Когда какая-либо система имеет дело с банковскими картами, она должна соответствовать глобальному стандарту безопасности данных индустрии платежных карт (PCI DSS). В США Акт о мобильности и подотчетности медицинского страхования (HIPAA) представляет собой набор стандартов, созданных для защиты охраняемой законом информации о состоянии здоровья (PHI) регулирующими поставщиками медицинских услуг. Общий регламент защиты персональных данных (GDPR) – это постановление Европейского союза, затрагивающее любую компанию, работающую с данными граждан Европы.

Независимо от причины, по которой существуют такие регламенты или стандарты, они могут влиять на данные, которыми манипулирует ваш API. И это не только функциональный вопрос; также существуют передовые технические рекомендации, которым должны следовать разработчики и проектировщики API. Например, обычно не рекомендуется возвращать последовательные ключи баз данных, которые дадут подсказки, касающиеся важных данных, например сколько клиентов в вашей компании; такие данные можно использовать, чтобы попытаться получить доступ к данным, принадлежащим другим лицам (хотя, если ваша реализация отлично справляется с управлением доступа, это не должно быть проблемой).

**ПРИМЕЧАНИЕ.** Всегда консультируйтесь со своим CISO (главный директор по информационной безопасности), DPO (главный специалист по защите данных), CDO (главный директор по обработке и анализу данных) или юридическим отделом, чтобы убедиться, какие данные следует считать конфиденциальными.

Как только вы определите, какие данные являются конфиденциальными, второй этап – выбор соответствующих представлений для этих данных. Напомню, что такая адаптация – это *проектирование API*. Поэтому нам необходимо сосредоточиться на том, что потребители могут делать с помощью этого API, а не на самих данных.

То, как представления будут адаптированы, во многом зависит от того, как будут использоваться данные. В случае использования дебетовой карты цель состояла в том, чтобы предоставить только подробную информацию о счете и связанных с ним картах, чтобы пользователи не могли выполнить платеж. В этом случае имеет смысл не указывать полный номер карты и ее CVV. Наличие такого ориентированного на цели мышления упростит адаптацию. На рис. 8.16 показаны четыре метода, которые можно использовать для создания безопасных представлений.

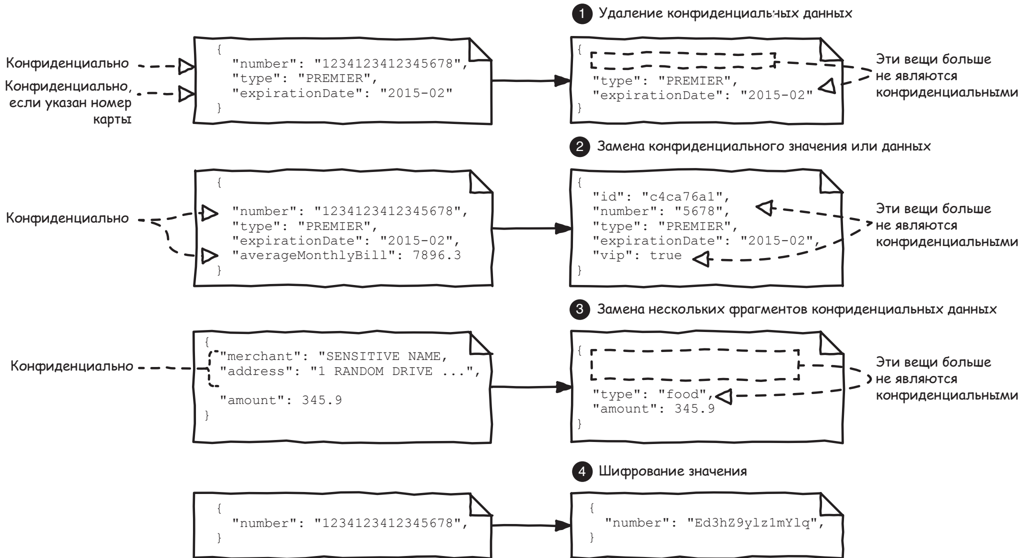



**Рис. 8.16. Адаптация представлений конфиденциальных данных, чтобы сделать их неконфиденциальными**

Первый метод прост: вы удаляете любые конфиденциальные данные, которые не требуются (1), что может иметь хороший побочный эффект: некоторые конфиденциальные данные, связанные с удаленными данными, могут стать неконфиденциальными. Например, на рис. 8.16 при удалении конфиденциального свойства `number` (номера карты) свойство `expirationDate` становится неконфиденциальным.

Если конфиденциальные данные нельзя удалить, можно заменить их неконфиденциальной адаптацией, как мы это сделали с номером карты (2). Здесь значение свойства `number` усекается, чтобы сделать его неконфиденциальным. Кроме того, свойство `averageMonthlyBill` заменено неконфиденциальным флагом `vip`, значение которого является результатом `someFunction(averageMonthlyBill)`. В этом случае процесс ориентированной на цели разработки, вероятно, решил бы эту проблему с самого начала.

Иногда бывает полезно сочетать замену значения и данных. Например, если конфиденциальные данные используются в качестве идентификатора, возможно, будет более практично просто создать новый не имеющий смысла идентификатор для идентификации ресурса, вместо

того чтобы использовать значимый, но конфиденциальный ID. Усеченное число, такое как 5678, может быть полезно, для того чтобы продемонстрировать его конечным пользователям, но не для использования в качестве идентификатора ресурса. Путь к ресурсу `/cards/5678`, вероятно, будет уникальным в контексте клиента, но, возможно, не в более широком контексте консультанта. Следовательно, представление карты может получить пользу от совершенно нового идентификатора, содержащего непрозрачный ID, такой как c4ca76a1. Получившийся в итоге путь к ресурсу `/cards/c4ca76a1` будет уникальным в обоих контекстах; и, в зависимости от ваших потребностей, этот новый идентификатор может быть заменой свойству `number` или дополнением к представлению карты. Обратите внимание, что, как и удаление конфиденциальных данных, их замена может сделать другие данные неконфиденциальными. Здесь свойство `expirationDate` становится неконфиденциальным после замены `number`.

Замена данных может применяться к нескольким свойствам (3). Использование вместо набора точных и конфиденциальных значений более нечеткого, но все же значимого набора также может помочь. В методе, приведенном на рис. 8.16, конфиденциальные свойства `merchant` и `address` транзакции по карте объединяются в новое неконфиденциальное свойство `type`, значение которого является результатом `someFunction(merchant, address)`.

В крайнем случае, если конфиденциальные данные действительно нужны как они есть и шифрования канала обмена данными недостаточно, можно зашифровать значения (4). В примере на рис. 8.16 значение `number`, равное `1234123412345678`, зашифровано как `Ed3hZ9ylz1mYlq`. Но у этой техники есть свои недостатки. Примечательно, что потребители должны будут расшифровать зашифрованные данные, чтобы использовать их. Когда дело доходит до этого, возможно, будет проще и эффективнее шифровать все сообщения, передаваемые по защищенному соединению, установленному между потребителем и поставщиком. Самым безопасным вариантом будет шифровать данные специально для каждого потребителя и предоставлять каждому из них правильный ключ для расшифровки данных.

Все это работа людей, управляющих уровнем безопасности и реализующих API. Но, будучи проектировщиком API, вы должны убедиться, что все эти средства безопасности удобны для потребителя. Общаясь со специалистами по безопасности и используя методы, которым вы научились, чтобы проектировать API, вы должны иметь возможность создавать безопасные представления. Но не только данные, которыми обмениваются через API, могут быть конфиденциальными – некоторые цели API также могут таковыми.

### 8.4.2 Обработка конфиденциальных целей

Есть два вида конфиденциальных целей: те, которые манипулируют конфиденциальными данными, и те, которые инициируют действия

с особо важными последствиями. Как и в случае с конфиденциальными данными, независимо от того, почему цель считается конфиденциальной, первый вопрос, который вы себе задаете, должен звучать так: «Действительно ли эта цель так необходима?» Если нет, то, чтобы сохранить дизайн API безопасным, не включайте его в API – это самый простой способ. Но это не всегда возможно.

Допустим, по очень веской причине банковский API должен предоставить конфиденциальную информацию о карте (такую как номер, CVV, срок действия и имя владельца) некоторым потребителям или конечным пользователям. Было бы целесообразно обеспечить строгий контроль доступа к этим данным. Как показано на рис. 8.17, этот вопрос можно решить четырьмя различными способами.

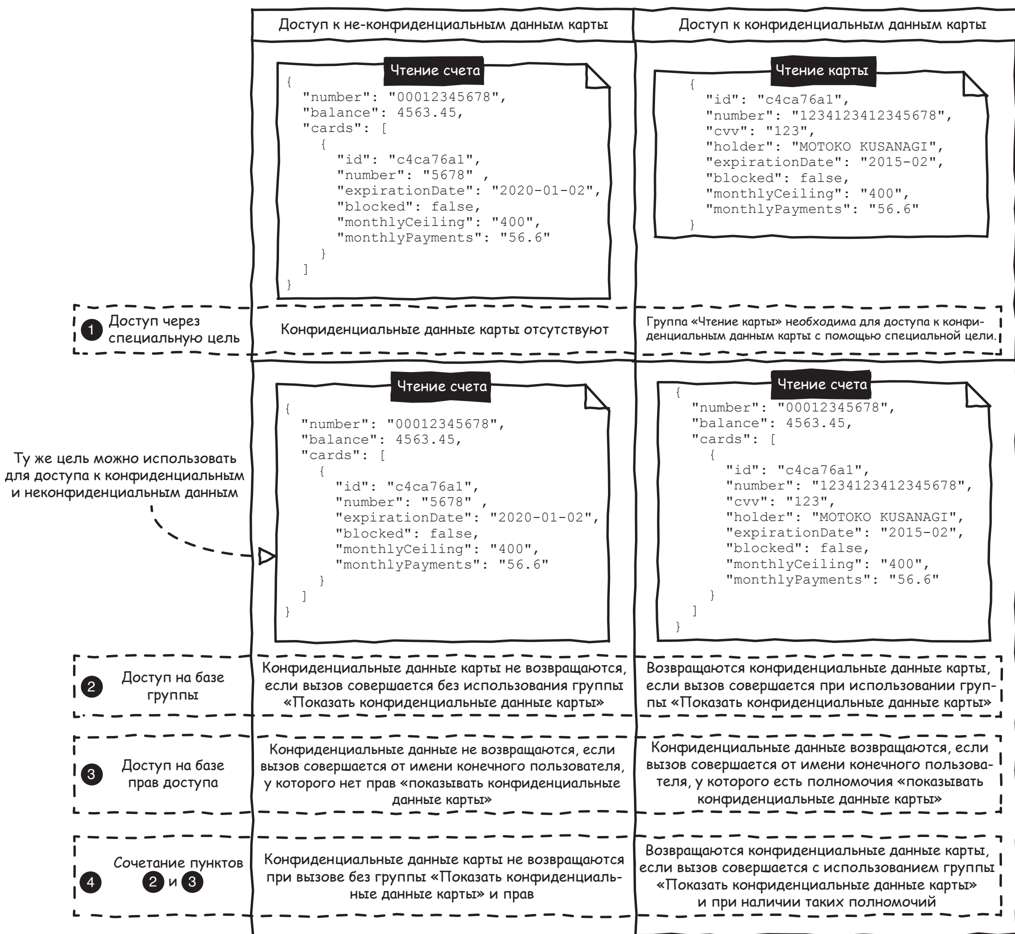



**Рис. 8.17.** Управление доступом к целям, раскрывающим доступ к конфиденциальным данным

Первым вариантом (1) может быть создание отдельной цели «Прочитать карту», обеспечивающей доступ к конфиденциальным данным карты. Эту цель можно защитить с помощью определенной группы «Чтение

карт», предоставляемой только тем потребителям, которые действительно в ней нуждаются. Цель «Прочитать счет» может остаться неизменной и по-прежнему предоставлять неконфиденциальную версию данных карты.

Второй вариант (2) может заключаться в использовании группы, чтобы инициировать возврат конфиденциальных данных. Цель «Прочитать счет» будет возвращать неконфиденциальные представления данных карты по умолчанию, за исключением потребителей, которым предоставлена группа показа конфиденциальных данных карты. Реализация этих целей будет возвращать необработанные конфиденциальные данные карты вместо адаптированных неконфиденциальных.

В первых двух вариантах, если конечные пользователи участвуют, они смогут видеть конфиденциальные данные при использовании потребителя с нужной группой. Это может не соответствовать некоторым требованиям безопасности. Следовательно, в качестве третьего варианта (3) можно было бы сделать то же самое, обрабатывая выбор возврата конфиденциального или неконфиденциального представления на основе прав доступа или ролей конечного пользователя, прикрепленных к токену доступа.

Но использование прав доступа только для конечных пользователей будет означать, что любой потребитель, которому разрешено использовать API от имени пользователей, будет иметь доступ к конфиденциальным данным. Если это проблема, можно использовать четвертый вариант (4), который заключается в объединении управления доступом со стороны потребителя и конечного пользователя, чтобы гарантировать, что доступ к конфиденциальным данным возможен только тогда, когда конечный пользователь с соответствующими правами использует потребителя с нужной группой.

Выбор подходящего для использования варианта зависит от ограничений безопасности, а также от опыта разработчика. Как и при разработке представлений безопасных данных, вам нужно будет побеседовать с сотрудниками службы безопасности вашей организации, чтобы узнать, какие варианты возможны. Кроме того, с точки зрения потребительского опыта, если между конфиденциальными и неконфиденциальными версиями данных существует большой разрыв, будет лучше предоставить специальные цели для доступа к конфиденциальным данным, вместо того чтобы модулировать возвращаемую версию на основе групп или прав.

Но цели могут быть конфиденциальными, даже если они не манипулируют конфиденциальными данными. Давайте считать само собой разумеющимся, что в нашем банковском API мы работаем над безопасным представлением данных карты, и ни одно из ее свойств не считается конфиденциальным. Помимо основных данных карты, это представление содержит флаг `blocked` и сумму, указанную в свойстве `monthlyCeiling`. Эти два свойства могут быть обновлены конечными пользователями или, точнее, потребителями от их имени. Можно было бы добавить цель «Обновить карту», позволяющую обновить эти дан-

ные. Но даже если обновление порога платежей, при превышении которого отправляется предупреждение, не является особо важным действием, таковым является блокировка карты. Поэтому было бы разумно убедиться, что блокировка карты выполняется только конечными пользователями или потребителями, которым разрешено это делать.

Для этого можно использовать те же параметры, что и для работы с конфиденциальными данными. Мы могли бы воспользоваться правами конечных пользователей и проверить реализацию, чтобы посмотреть, разрешено ли пользователю обновлять флаг `blocked` при использовании цели «Обновить карту». Также мы могли бы полагаться на группу «Блокировка карты», которая позволила бы потребителю делать то же самое при использовании вышеупомянутой цели. И, очевидно, мы могли бы смешать эти два варианта. Все это будет работать, но смешивание обновлений конфиденциальных и неконфиденциальных данных в одной цели может усложнить API, особенно если обновление первых требует дополнительных мер безопасности, таких как многофакторная аутентификация конечных пользователей. В этом случае свойство `monthlyCeiling` может обновляться сразу, а свойство `blocked` – только тогда, когда его проверяет пользователь. Вероятно, было бы лучше создать отдельную цель «Блокировать карту» для обработки этого обновления. Если нам сделать шаг назад и еще поразмыслить над тем, что может делать API, и меньше думать о самих данных, такая цель имеет смысл.

Таким образом, обработка конфиденциальных целей требует, чтобы мы определили, манипулируют ли они конфиденциальными данными или инициируют особо важные действия. Как только это будет сделано и мы действительно будем уверены, что нам это нужно, мы можем воспользоваться группой или правами доступа, чтобы гарантировать, что только авторизованные потребители и конечные пользователи могут получить доступ к этим целям. Но использование одних только групп или прав доступа может быть недостаточным.

Иногда лучше адаптировать контракт интерфейса, чтобы предоставить отдельные, четко определенные и детализированные цели для обработки особо важных частей (данных или действий). Такая адаптация облегчает управление доступом и делает API простым для понимания и использования. Но что произойдет, если потребители попытаются сделать то, что им запрещено? Какое сообщение они получат в ответ? И должны ли проектировщики API также заботиться о безопасности при проектировании других видов сообщений?

### 8.4.3 Проектирование безопасных сообщений об ошибках

В разделе 5.2 вы узнали, как спроектировать исчерпывающие и информативные сообщения об ошибках, которые помогают потребителям самостоятельно решать проблемы. Там мы определили два типа ошибок: вызванные неправильно сформированным запросом и нарушениями бизнес-правила. Здесь нам нужен новый тип, чтобы четко обозначить ошибки, связанные с безопасностью, и мы можем использовать тот же тип представления – сообщение и код. Что может произойти, когда по-

требитель инициирует ошибку, связанную с безопасностью, при вызове банковского API, скажем, путем отправки запроса `GET /cards/c4ca76a1`?

Если потребитель отправляет запрос без предоставления токена или предоставляет недействительный токен, сервер банковского API способен вернуть в ответ `401 Unauthorized`. Этот ответ может содержать тело с таким сообщением, как `Missing or invalid token`. Если тот же потребитель повторяет свой запрос с действительным токеном, но ему не был предоставлен доступ к группе «Чтение карты», сервер может вернуть в ответ `403 Forbidden`, сообщение `Consumer has not been granted the "read card" scope message` и код SCOPE. А если потребитель попытается сделать это снова, после того как разработчик обновит конфигурацию, чтобы добавить недостающую группу, сервер может снова вернуть в ответ `403 Forbidden` и сообщение `End user is not allowed to access this card` с кодом `PERMISSIONS`. Последний ответ, очевидно, означает, что конечный пользователь не имеет прав на чтение этой конкретной карты.

В некоторых случаях эту последнюю ошибку можно считать *утечкой информации*. В самом деле, явное упоминание о том, что у пользователя нет доступа к карте `c4ca76a1`, косвенно подтверждает, что эта карта действительно существует. Чтобы избежать подобной утечки, сервер банковского API может сказать, что карта не существует в контексте конечного пользователя. Он может сделать это, вернув в ответ `404 Not Found` и сообщение `Card does not exist`, так же как это происходит, когда потребитель запрашивает карту, которой на самом деле не существует.

Также нужно учитывать риск утечки информации при проектировании некорректного запроса или сообщения о функциональной ошибке. Например, потребитель может попытаться обновить карту с помощью запроса `PATCH /cards/c4ca76a1`, в котором используется безопасный идентификатор карты. Если потребитель предоставляет недопустимое значение свойства `monthlyCeiling` для обновления этой карты, вернуть сообщение, содержащее конфиденциальную информацию, такую как полный номер карты, например `Impossible to update the 1234123412341234 card`, было бы ужасной идеей.

Есть еще один тип ошибок, который мы еще не обсуждали и который подвержен критической утечке информации: непредвиденные. Например, старая добрая ошибка исключения `java.lang.NullPointerException` (пытающаяся `что-то.сделать()`, когда `что-то` является `null` в Java) или довольно раздражающее сообщение `Unable to extend table in tablespace` (когда недостаточно дискового пространства для расширения локальной таблицы базы данных, чтобы добавить данные). Очевидно, что таких ошибок никогда не происходит, потому что все полностью протестировано, проверено и автоматизировано...до тех пор, пока они не произойдут, обычно в самый неподходящий момент.

При использовании протокола HTTP такие ошибки обозначаются кодом состояния класса `5XX`. Обычно это – `500 Internal Server Error`. В то время как класс `4XX` означает, что это вина потребителя, `5XX` означает, что виноват поставщик. При возникновении таких непредвиденных

ошибок сервера реализация ни за что не должна предоставлять подробную информацию о техническом стеке, стоящем за API. Вы можете предоставить идентификатор ошибки для дальнейшего изучения, но будьте осторожны, чтобы не предоставить информацию, которая может дать подсказку о том, что на самом деле работает за интерфейсом. Таким образом, в таких ошибках не может быть никакой трассировки стека и подробных описывающих ошибки версий программного обеспечения, адресов серверов и т. п. Таблица 8.1 суммирует различные типы ошибок, которые вы видели до сих пор.

**Таблица 8.1. Варианты использования сообщений об ошибках**

| Тип ошибки | Случай использования | Код состояния HTTP |
|---|---|---|
| Ошибка безопасности | Учетные данные отсутствуют либо они недействительны | `401 Unauthorized` |
| Ошибка безопасности | Недопустимые группы | `403 Forbidden` |
| Ошибка безопасности | Недействительные права доступа | `403 Forbidden or 404 Not Found` |
| Некорректный запрос | Неизвестный ресурс (неверный параметр пути) | `404 Not Found` |
| Некорректный запрос | Отсутствует обязательный параметр либо он недействителен | `400 Bad Request` |
| Функциональная ошибка | Нарушение бизнес-правила | `400 Bad Request or 403 Forbidden` |
| Техническая ошибка | Неожиданная ошибка сервера | `500 Internal Server` |

Как видите, сообщения об ошибках безопасности напоминают сообщения для других типов ошибок. При проектировании REST API вам просто нужно использовать соответствующий код состояния HTTP для этих ошибок, например 401 или 403. Но не забывайте, что нужно проявлять осторожность и следить за тем, какую информацию вы предоставляете с помощью сообщения об ошибке безопасности или технической ошибке (или любого другого типа), чтобы избежать непреднамеренного предоставления доступа к конфиденциальной информации. Мы почти закончили, но есть еще одна тема, о которой нужно знать, чтобы убедиться, что ваш API обрабатывает конфиденциальные материалы надлежащим образом.

### 8.4.4 Выявление проблем, связанных с архитектурой и протоколом

Насколько мы должны доверять архитектуре и протоколу, которые будут поддерживать проектируемые нами API? Не особо, если мы мало что о них знаем. На рис. 8.18 показана базовая (и некорректная) архитектура API, которая может использоваться для банковского REST API. В этом

примере показано, что безопасное соединение между потребителем и поставщиком не всегда может быть таким безопасным, как мы думаем.

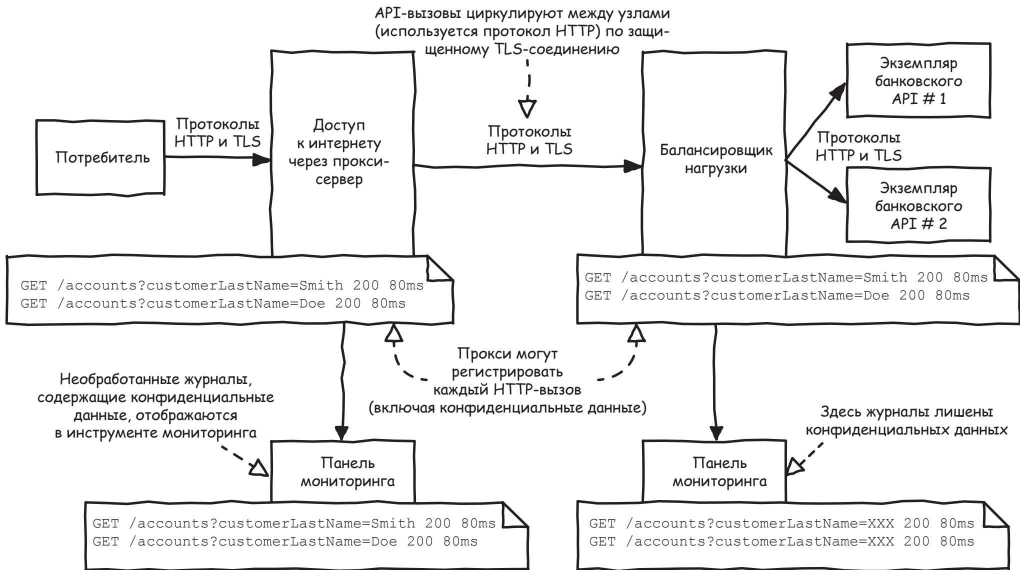



**Рис. 8.18.** Не очень безопасное соединение между потребителем и поставщиком

В этой базовой архитектуре потребитель проходит через прокси-сервер, чтобы получить доступ в интернет и выполнить вызовы к банковскому API по протоколу HTTP. Доступ к банковскому API не предоставляется напрямую в интернете; есть еще один прокси-сервер, балансировщик нагрузки HTTP, который распределяет запросы по различным экземплярам серверного приложения Banking API. Обмен данными между узлами защищен с использованием протокола TLS.

Кажется, что все абсолютно безопасно, но если присмотреться, то можно увидеть, что балансировщик нагрузки регистрирует некие данные (метод HTTP; URL-адрес, включая параметры запроса; код состояния HTTP; и время ответа). Содержимое этих файлов журнала отправляется в инструмент мониторинга, который можно использовать для создания блестящих полезных информационных панелей, показывающих, как используется банковский API и как он ведет себя. Это означает, что любой, кто имеет доступ к журналам балансировщика нагрузки или инструменту мониторинга, может увидеть данные, содержащиеся в пути к ресурсу, и его параметры запроса.

Банковская компания, конечно же, очень строга, когда дело касается безопасности. Во всех этих журналах нет конфиденциальных данных. Однако компания не контролирует, что происходит на прокси-сервере, используемом потребителем, который также может записывать в журнал вызовы по протоколу HTTP. Очевидно, что это забота потребителя. Но банковский API может быть спроектирован таким образом, чтобы гарантировать, что, если URL-адреса записываются в журнал, они не содержат никакой конфиденциальной информации.

Например, если список счетов можно отфильтровать по имени клиента с помощью запроса типа `GET /accounts?customerLastName=Smith`, это проблема. Имя является конфиденциальной информацией, и параметр запроса `customerLastName` будет отслеживаться в каждом HTTP-логе по сети между потребителем и поставщиком. Чтобы избежать этой потенциальной утечки данных, было бы безопаснее предложить запрос `POST /accounts/search`, тело которого содержит параметры поиска.

Если вы проектируете REST HTTP API, постарайтесь не помещать конфиденциальную информацию в параметры пути или параметры запроса, поскольку они могут записываться в журнал. Когда речь идет о протоколе, используемом для обмена данными, и архитектуре, созданной для API, всегда проверяйте, нет ли возможных утечек данных. Если есть потенциальные утечки, с помощью которых можно получить доступ к конфиденциальным данным, вы должны соответствующим образом адаптировать дизайн API, чтобы предотвратить это.

В следующей главе вы узнаете, что для эволюции API требуется дополнительная осторожность, чтобы избежать провоцирующих ошибок на стороне потребителя, и как минимизировать этот риск с нуля при проектировании API.

## Резюме

- Проектировщики API вносят большой вклад в безопасность API, сводя к минимуму поверхность атаки.
- API должен предоставлять доступ только к тому и запрашивать только то, что действительно нужно.
- Потребителям должно быть разрешено использовать только то, что им действительно нужно.
- Для обеспечения безопасности проектирование API должно выполняться с точки зрения пользователя с учетом того, какие данные необходимы для управления доступом.
- Конфиденциальные данные и цели охватывают широкий спектр; именно то, что следует считать конфиденциальным, может быть неочевидным и должно определяться с помощью технических специалистов, специалистов по безопасности, бизнесу и профессиональных юристов.
- Проектировщики API должны знать о потенциальных утечках, возникающих из-за базового протокола или архитектуры, чтобы полностью защитить проектирование API.

# 9 Изменение дизайна API

**В этой главе мы рассмотрим:**

- проектирование модификаций во избежание критических изменений;
- версионирование API для управления критическими изменениями;
- проектирование расширяемых API для ограничения критических изменений.

В предыдущих главах вы узнали, как проектировать API, которые предоставляют функции или цели, имеющие смысл для их пользователей. Вы также узнали, как создавать удобные для пользователя и безопасные представления этих целей. После того как был проделан весь этот труд, можно ли считать, что на этом работа создателя API подошла к концу? Вовсе нет! Это новое начало.

API – живое существо, которое неизбежно будет эволюционировать, возможно, для того чтобы предоставлять новые функции или улучшать уже существующие. Для проектирования таких изменений вы можете повторно использовать те же навыки, которые уже изучили к этому моменту, но это требует дополнительного внимания.

За эти годы я купил несколько книжных шкафов Ikea Billy для хранения книг, комиксов, компакт-дисков, пластинок и много другого. Стандартный книжный шкаф Billy поставляется с четырьмя подвижными полками, которые вы можете разместить по своему усмотрению благодаря множеству отверстий, просверленных с обеих сторон. Вам просто нужно разместить четыре колышка на нужной высоте, поставить на них полку – и

готово. Если книжный шкаф используется для хранения небольших предметов, таких как компакт-диски или книги в мягкой обложке, при использовании только четырех подвижных полок может оставаться много пустого пространства. К счастью, можно отдельно приобрести дополнительные полки, что позволяет использовать пустое место для хранения более мелких предметов. Но в последний раз, покупая дополнительные полки для шкафа, владельцем которого я был в течение достаточно долгого времени, я столкнулся с неприятным сюрпризом – колышки были слишком маленькими, чтобы встать в отверстия моего старого доброго шкафа Billy.

В системе Billy версии 2.0 используются колышки меньшего диаметра, которые несовместимы с отверстиями предыдущей версии. Я не знал об этом, пока не попытался использовать новые колышки; когда я покупал дополнительные полки, это нигде не было указано. Досадная ситуация. Такие *критические изменения* могут происходить и при проектировании API.

Что может произойти, если поставщик банковского API решил, что вместе с информацией о банковском счете теперь должна возвращаться валюта баланса, чтобы обеспечить поддержку различных валют? На рис. 9.1 показано, что развитие банковского API таким образом может привести к сценарию, эквивалентному изменению диаметра колышков для шкафа.

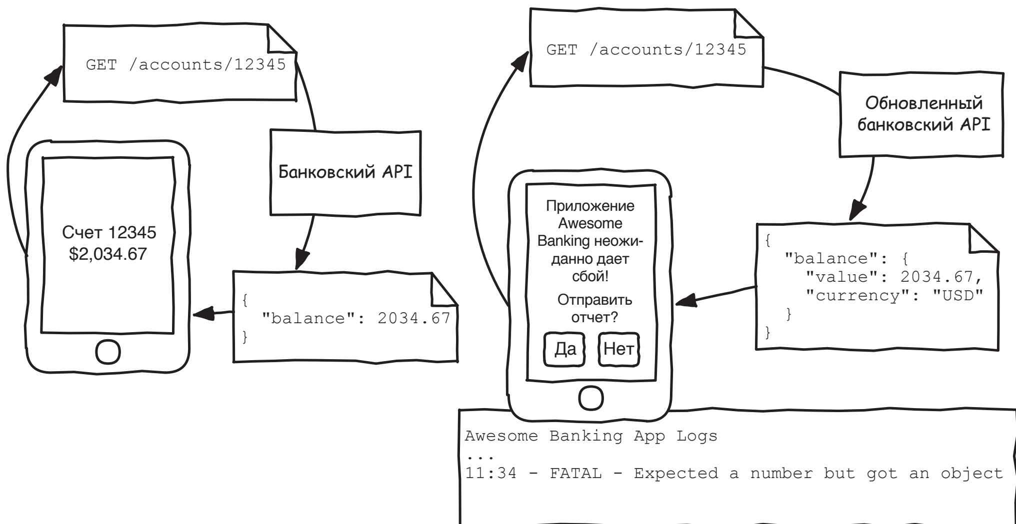



**Рис. 9.1.** Потребитель сталкивается с критическим изменением после обновления банковского API

В этом сценарии свойство `balance`, которое было числом, теперь является объектом, содержащим значение (прежний баланс) и его валюту (строка кода валюты согласно стандарту ISO 4217). Хотя и умно использовать кода валюты ISO 4217 (как вы узнали в разделе 6.1.4), менять баланс таким способом – определенно плохая идея. Приложение Awesome Banking аварийно завершает работу при парсинге возвращаемых данных, поскольку ожидает, что свойство `balance` будет числом, а не объектом. Разработчики, отвечающие за написание кода этого потребителя,

обычно перехватывают такие ошибки, избегая подобных сбоев, но ошибки все равно будут препятствовать нормальной работе потребительского приложения. Чтобы решить эту проблему, нужно обновить мобильное приложение.

*Критическое изменение* – это изменение, которое вызовет проблемы у потребителей, если они не обновят свой код. В большинстве случаев нельзя синхронизировать обновление API со всеми обновлениями его потребителей, поэтому, безусловно, важно стараться избегать таких критических изменений или хотя бы знать о них при проектировании изменений API. Мы можем тщательно спроектировать их, чтобы избежать внесения проблемных изменений. Мы даже можем спроектировать API с нуля, чтобы предотвратить их. Но, независимо от того, насколько тщательно мы проектируем наши API и их развитие, рано или поздно критические изменения неизбежны.

Будучи проектировщиками API, мы также должны знать о невидимой стороне контракта API – обо всех наблюдаемых действиях, явно не описанных в контракте интерфейса, которые могут незаметно развиваться и вызывать совершенно неожиданные критические изменения.

Чтобы справиться с этими ситуациями, когда они возникают, разумно знать, как управлять версиями наших API. Мы рассмотрим все эти темы в этой главе. Начнем с изучения того, как (тщательно) проектировать изменения.

## 9.1 Проектирование изменений API

Эволюция банковского API, описанная во введении к этой главе, проиллюстрировала возможный способ внесения критических изменений. Потребитель потерпел сбой из-за измененной структуры данных, которую стало невозможно анализировать (число стало объектом). Но это не единственный способ внести критические изменения. Хотя некоторые из них довольно очевидны, как это изменение структуры данных, другие более коварны, как например, изменение возможных значений свойства. Кроме того, последствия не всегда могут быть такими очевидными, потому что изменения могут не вызвать видимую ошибку на стороне потребителя. Критические изменения могут даже повлиять на поставщика. Представьте себе, каковы могут быть последствия как для потребителей, так и для поставщика, если значения суммы в долларах в банковском API будут заменены значениями в центах, особенно для цели «Перевести деньги».

Любая модификация контракта интерфейса API, которую формально можно описать с использованием формата описания API или текстовой документации API, может привести к критическим изменениям. Это относится к выходным и входным данным, параметрам, состояниям ответов или ошибкам, целям и потокам и безопасности. Поэтому знание того, как избежать появления критических изменений, когда это возможно, и их изящной обработки в противном случае, крайне важно для проектировщика API.

### 9.1.1 Избегая критических изменений в выходных данных

Банковский API предлагает цель «Перечислить транзакции», которая возвращает список транзакций для номера счета. В левой части рис. 9.2 показаны данные, возвращаемые для каждой транзакции. В правой части – переработанная версия, иллюстрирующая различные способы внесения критических изменений, которые вызовут проблемы у потребителей при получении списка транзакций по счету.

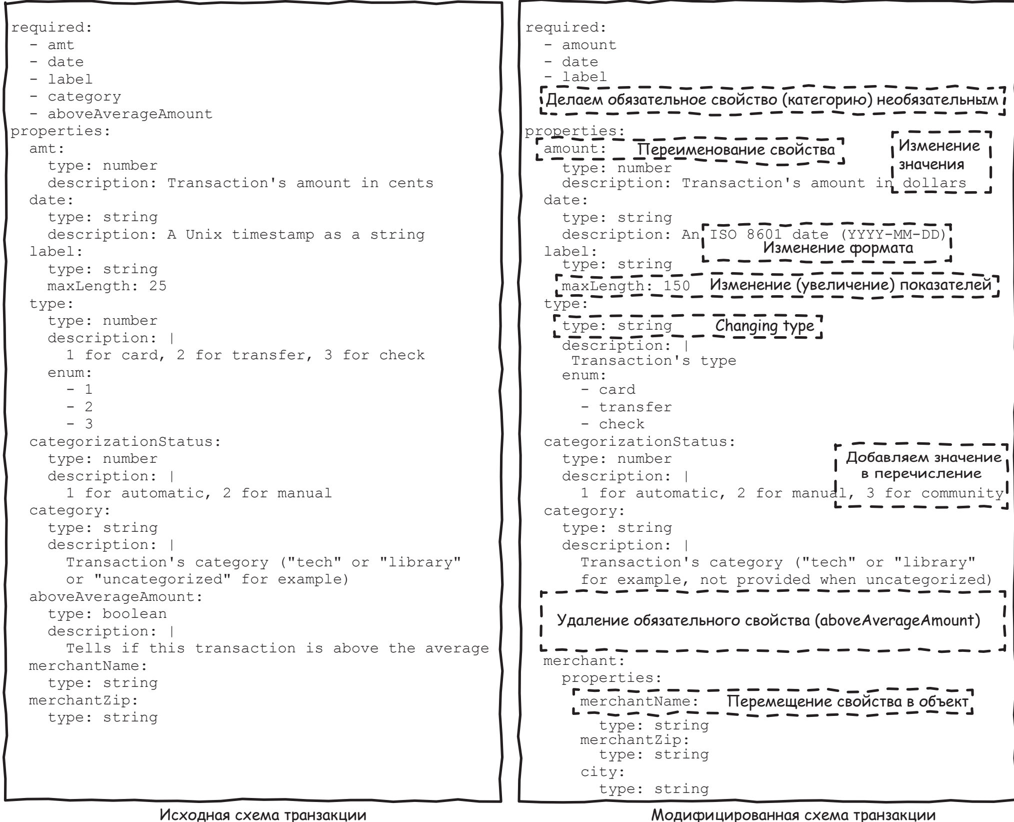



**Рис. 9.2. Как внести критические изменения в выходные данные цели «Перечислить транзакции»**

Новый проектировщик переименовал свойство `amt` в `amount`, чтобы сделать его более понятным. Исходя из того, что вы узнали, это хороший дизайн, но это изменение может оказать значительное влияние на потребителя. Версия приложения Awesome Banking для Android, которая не так хорошо написана, может привести к сбою при использовании знаменитой ошибки исключения `java.lang.NullPointerException`. Версия для iOS может просто показывать транзакции без сумм, вызывая раздражение у конечных пользователей. Более сложный потребитель Financial Statistics, который выполняет вычисления на основе суммы транзакции, может посчитать, что сумма каждой транзакции равна 0, и это может повредить его данные. Кроме того, перемещение свойств `merchantName`

и merchantZip в новую структуру merchant (поскольку было добавлено свойство city, а проектировщик, предположительно, прочитал раздел 7.1.1 этой книги) – также примеры критических изменений, вызывающих исключение при парсинге.

А как насчет свойства aboveAverageAmount, которое было удалено, возможно, потому что эта информация не считалась важной? Это тоже проблема? Определенно, потому что это свойство является обязательным. В первоначальной версии этой транзакции оно должно было предоставляться всегда, поэтому его удаление может привести к тем же проблемам, что и переименование свойства amt или перемещение merchantName.

Еще одна проблема связана с типом транзакции (type), которая представляет собой число, указывающее, идет ли речь о карте (1), переводе (2) или чеке (3). Теперь это – строка (потому что новый проектировщик знает, что читабельный код обычно лучше). Несмотря на благосклонность, такое изменение, вероятно, приведет к ошибке парсинга на стороне потребителя; и даже если этого не произойдет, интерпретация новых значений, вероятно, будет невозможна.

Переименование, перемещение или удаление свойств и изменение их типов являются очевидными способами внесения критических изменений в выходные данные. Но некоторые другие модификации более коварны.

Свойство category было обязательным, но его сделали необязательным. Потребители всегда получали его, а теперь, если этого не происходит, они могут столкнуться с теми же проблемами, что и в случае переименования свойства amt. Свойство date было строкой, и оно по-прежнему ей является, но формат изменился. Это была временная метка UNIX со строковым форматом. Хотя обычно оно было представлено числом, теперь это – дата в формате ISO 8601 (новый проектировщик решил исправить это). Опять же, это благотворное изменение, но оно приведет к ошибкам парсинга на стороне потребителя.

Изменение свойства label также может быть критическим изменением. Его максимальная длина была изменена с 25 до 150, возможно, из-за того, что основная банковская система, стоящая за банковским API, была обновлена, чтобы управлять целиковыми метками и прекратить усекать их. Если на стороне потребителя это значение хранится в старой доброй реляционной базе данных, где размер ее столбца определен как 25, хранить более длинные значения будет невозможно. Это коварные критические изменения, но есть и менее очевидные.

Посмотрите внимательнее на описания. В исходной версии в описании amt видно, что сумма транзакции указана в центах; но в новой версии это значение указано в долларах.

При получении значения свойства amt, например, 3034 (центы), потребители понимали, что это 30,34 долл. Теперь они получат значение 30,34 в долларах и будут воспринимать ее как 0,3034 долл. Это может спровоцировать панику на стороне потребителя.

А теперь, что менее критично, свойство categorizationStatus. Оно представляло собой числовой код, указывающий, как была классифици-

рована транзакция: 1 – автоматическая транзакция и 2 – ручная. В новой версии добавлено новое значение кода. Потребители не смогут интерпретировать значение 3 без обновления. И даже если бы этот код был удобочитаемым, такая модификация могла бы стать проблемой, потому что приложение, использующее API, не смогло бы его интерпретировать.

Есть множество разных способов внести критические изменения. В табл. 9.1 собраны различные типы модификаций и их последствия.

**Таблица 9.1. Критические изменения в выходных данных и их последствия**

| Модификация | Последствия |
|---|---|
| Переименование свойства | Варьируется в зависимости от реализации (недостающие данные в пользовательском интерфейсе, повреждение данных, сбой и т. д.) |
| Перемещение свойства | Варьируется в зависимости от реализации (недостающие данные в пользовательском интерфейсе, повреждение данных, сбой и т. д.) |
| Удаление обязательного свойства | Варьируется в зависимости от реализации (недостающие данные в пользовательском интерфейсе, повреждение данных, сбой и т. д.) |
| Обязательное свойство становится необязательным | Варьируется в зависимости от реализации (недостающие данные в пользовательском интерфейсе, повреждение данных, сбой и т. д.) |
| Изменение типа свойства | Ошибка парсинга |
| Изменение формата свойства | Ошибка парсинга |
| Изменение характеристик свойства (увеличение длины строки, диапазона чисел или количества элементов массива) | Варьируется в зависимости от реализации (ошибки базы данных и пр.) |
| Изменение значения свойства | Ожидайте худшего |
| Добавление значений в перечисления | Варьируется в зависимости от реализации (недостающие данные в пользовательском интерфейсе, повреждение данных, сбой и т. д.) |

Этот список может быть неполным на 100 %, но идею вы поняли. Как видите, изменение существующих элементов в выходных данных может вызвать более или менее очевидные критические изменения с более или менее значительными последствиями.

Теперь, когда мы знаем, как вносить критические изменения в выходные данные, давайте посмотрим, какие изменения можно выполнить безопасно. На рис. 9.3 показано обратно совместимое изменение схемы транзакции.

Информация о торговом городе была просто добавлена как `merchantCity` без изменения существующих свойств. Новый статус категоризации, который должен был указывать на то, что категоризация была автоматической, хотя и была основана на данных других клиен-

тов, обрабатывается новым булевым флагом `communityCategorization`. А замена кодовых номеров типа транзакции на удобочитаемые производится путем добавления нового свойства `typeLabel`. Потребителей не будут беспокоить эти новые элементы.

Еще одно изменение касается свойства `type`, которое было необязательным, а теперь стало обязательным. Вместо того чтобы *иногда* получать это свойство, потребители будут получать его *всегда*. В отличие от обязательного свойства, которое становится необязательным, это некритическое изменение. Нужно ли вносить такие изменения? Вероятно, нет. На самом деле это не особенно важно. Потребители могут продолжать использовать API без уведомления о типе транзакции.

Формат метки также был изменен обратно-совместимым способом (исключительно в целях иллюстрации): ее максимальная длина теперь составляет 25 вместо 100. В качестве еще одного примера некритического изменения можно удалить необязательное свойство `categorizationStatus`. В зависимости от того, как данные сериализуются (не все API используют JSON), это может вызвать некоторые проблемы, поэтому будет лучше оставить его и всегда возвращать нулевое значение.

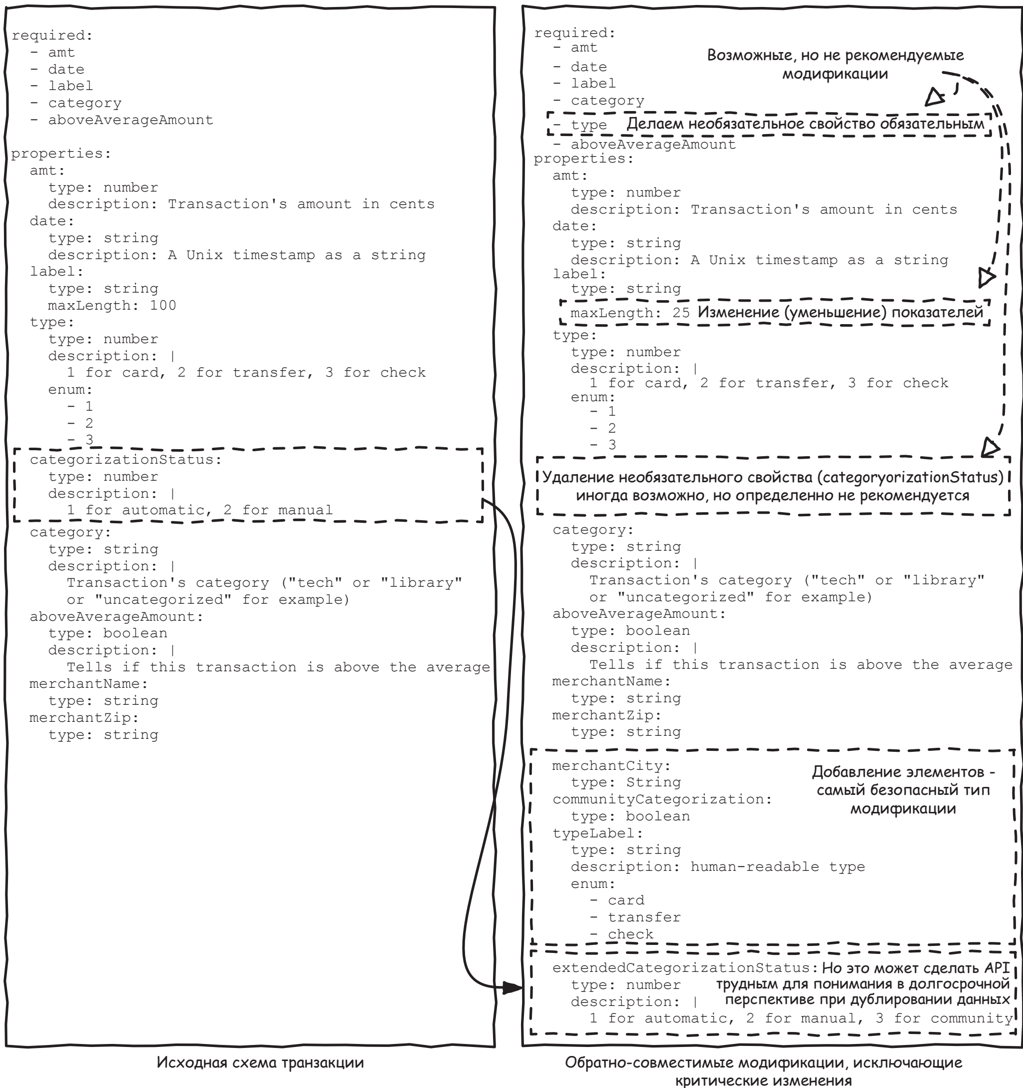



**Рис. 9.3.** Проектирование обратно-совместимых модификаций выходных данных

Обратите внимание, что некоторые изменения, которые должны были исправить плохой дизайн, такие как изменение имени свойства amt, выполнить нельзя. Это то, с чем приходится сталкиваться проектировщикам API. Как только потребители начинают использовать плохо спроектированный API, большую часть времени его невозможно исправить, не внося критических изменений.

Возможно, конечный результат - не самый подходящий вариант но, по крайней мере, он позволяет новому проектировщику вводить новые функции и частично исправлять некоторые ошибки проектирования, не нарушая код потребителей. Честно говоря, самый безопасный способ из-

менить выходные данные – просто-напросто добавить новые элементы. В случае с новыми функциями (такими как `merchantCity`) это довольно просто: всего лишь добавьте новые необходимые данные. Но когда дело доходит до незначительной модификации существующих вещей (например, значения статуса категоризации), найти решение будет сложнее. Здесь нет волшебного рецепта, но вы можете попробовать два подхода, как показано на рис. 9.3.

Во-первых, можно рассматривать это новое значение в качестве флага, как это было сделано на рис. 9.3, добавив логическое значение `communityCategorization`. Во-вторых, вы можете добавить новое свойство (скажем, `extendedCategoryStatus`), которое показывает те же данные, что и `categorizationStatus`, плюс новое состояние. Однако, если это было сделано несколько раз, полученный в итоге результат, включающий дублирование данных, может выглядеть неуклюже и затруднит понимание API.

А как насчет изменения входных данных и параметров? Это работает так же? Почти. И это тонкое различие важно знать. Давайте внесем критические изменения во входные данные цели «Перевести деньги», чтобы сопоставить это с тем, что вы узнали об изменениях выходных данных.

### *9.1.2 Как избежать критических изменений во входных данных и параметрах*

Входные данные, показанные слева на рис. 9.4, были немного изменены по сравнению с тем, с чем мы работали в предыдущих главах, чтобы проиллюстрировать различные (возможные) критические изменения. В правой части рисунка показаны различные способы, с помощью которых новый проектировщик, занимающийся доработкой API, может вносить критические изменения во входные данные.

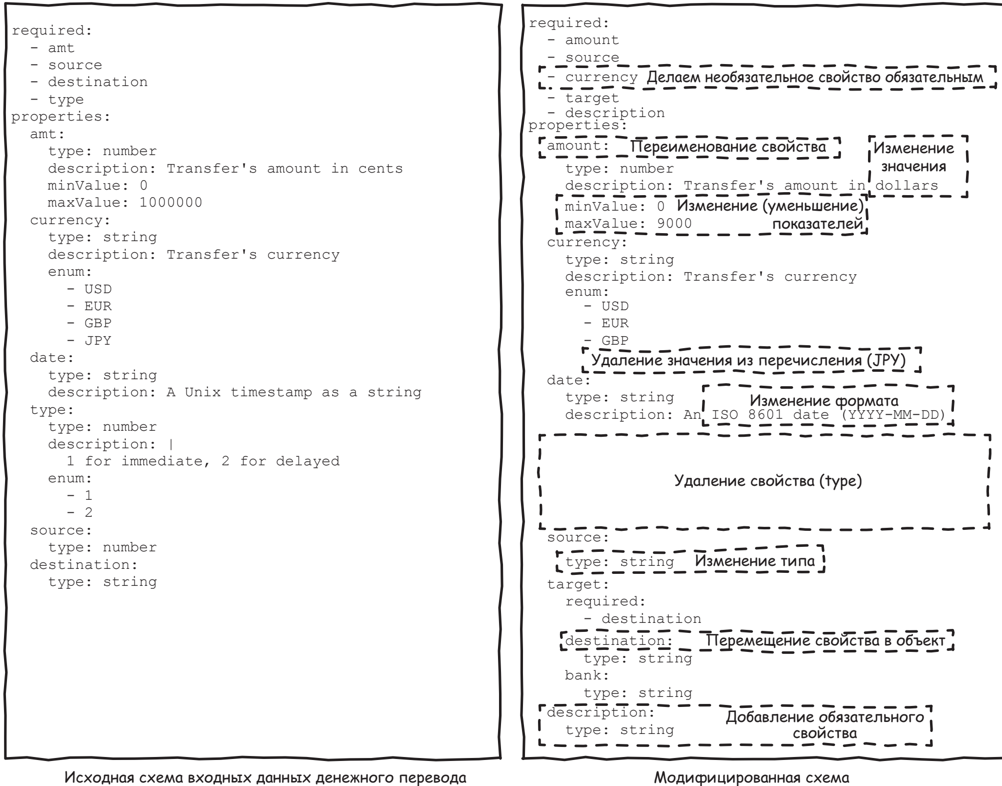



**Рис. 9.4 Как внести критические изменения во входные данные цели «Перевести деньги»**

Если проектировщик переименовывает свойство `amt` в `amount`, необновленный потребитель, отправляющий запрос на перевод денег с помощью `amt`, получит ошибку. В зависимости от того, как осуществлена реализация, эта ошибка может быть вызвана тем, что теперь `amt` – это *неожиданное* свойство или потому что новое свойство `amount` является *обязательным*. Для REST API это будет означать, что оно вернет код состояния `400 Bad Request`.

То же самое касается перемещения свойства (здесь мы перемещаем `destination` внутри `target`), изменения типа свойства (меняем `source` с числа на строку) и изменения форматов в целом (изменение формата даты и диапазона суммы).

Как и в случае с выходными данными, изменение значения свойства является бесшумным критическим изменением. Ранее сумма была указана в центах, а теперь – в долларах; следовательно, потребитель, отправляющий денежный перевод в размере 8 000 центов, вместо этого инициирует перевод в размере 8 000 долл. Это ужасный побочный эффект, и банк, предоставляющий API, должен будет вернуть деньги, потому что ошибка на его стороне.

Эти критические изменения имеют одинаковый эффект как в случае использования входных, так и выходных данных, но это не тот случай,

когда необходимо удалить обязательное свойство, сделать обязательное свойство необязательным или добавить значения в перечисления. Удаление обязательного свойства `type` является критическим изменением, которое приведет к появлению ошибки `unexpected property`. Если бы свойство `type` было необязательным, было бы то же самое. Когда обязательное свойство становится необязательным, это не имеет абсолютно никаких последствий для поставщика или потребителей, но, когда происходит наоборот (например, с валютой), это приведет к отсутствию ошибок, указывающих на отсутствие обязательных свойств. Добавление значений в перечисления также не имеет последствий, а вот удаление значений приведет к ошибкам, указывающим на наличие недопустимых значений.

Значит ли это, что, как и в случае с выходными данными, добавление данных является самым безопасным способом изменения входных данных? Не совсем. Добавление обязательного свойства имеет те же последствия, что и превращение необязательного свойства в обязательное: API вернет ошибку `missing mandatory property`.

Критические изменения немного отличаются, когда дело доходит до входных и выходных данных. В табл. 9.2 собраны возможные типы критических изменений во входных данных и их влияние, а также приводится сравнение их с эффектами тех же или аналогичных изменений в выходных данных.

**Таблица 9.2. Критические изменения во входных данных и их последствия**

| Модификация | Последствия | Влияние на входные данные по сравнению с выходными |
|---|---|---|
| Переименование свойства | Ошибка API | Идентично |
| Перемещение свойства | Ошибка API | Идентично |
| Удаление обязательного или необязательного свойства | Ошибка API | Идентично |
| Необязательное свойство становится обязательным | Ошибка API | Противоположный эффект (то же, что сделать обязательное свойство необязательным) |
| Изменение типа свойства | Ошибка API | Идентично |
| Изменение формата свойства | Ошибка API | Идентично |
| Изменение характеристик свойства (уменьшение длины строки, диапазона чисел или количества элементов массива) | Ошибка API | Противоположный эффект (то же самое, что и увеличение) |
| Изменение значения свойства | Ожидайте худшего (в основном влияет на поставщика) | Противоположный эффект (влияет в основном на поставщика) |

| Модификация | Последствия | Влияние на входные данные по сравнению с выходными |
|---|---|---|
| Удаление значений из перечислений | Ошибка API | Противоположный эффект (то же, что и добавление значений) |
| Добавление обязательного свойства | Ошибка API | Нет ошибки (некритическое изменение) |

Итак, как же можно изменить входные данные обратно-совместимым способом? Давайте проанализируем рис. 9.5, чтобы выяснить это.

```
required:
  - amt
  - source
  - destination
  - type
properties:
  amt:
    type: number
    description: Transfer's amount in cents
    minValue: 0
    maxValue: 1000000
  currency:
    type: string
    description: Transfer's currency
    enum:
      - USD
      - EUR
      - GBP

  date:
    type: string
    description: A Unix timestamp as a string
  type:
    type: number
    description: |
      1 for immediate, 2 for delayed
    enum:
      - 1
      - 2
  source:
    type: number
  destination:
    type: string
```

Исходная схема входных данных денежного перевода

```
required:
  - amt
  - source
  - destination
properties:
  amt:
    type: number
    description: Transfer's amount in cents
    minValue: 0
    maxValue: 1500000
  currency:
    type: string
    description: Transfer's currency
    enum:
      - USD
      - EUR
      - GBP
      - JPY
  date:
    type: string
    description: A Unix timestamp as a string
  type:
    type: number
    description: |
      1 for immediate, 2 for delayed
    enum:
      - 1
      - 2
  source:
    type: number
  destination:
    type: string
  destinationBank:
    required:
      - name
    properties:
      name:
        type: string
      country:
        type: string
```



Модифицированная схема

**Рис. 9.5. Проектирование обратно-совместимых модификаций входных данных**

Возможно, вы уже догадались, что самый безопасный способ – добавить только необязательные свойства. Свойство `destinationBank` является необязательным, поэтому, если потребители не предоставят его в своих запросах, это не приведет к ошибке. Но обратите внимание, что свойство `name` внутри `destinationBank` является обязательным. И в самом деле, если добавленные свойства являются объектами, не имеет значения, являются ли их свойства обязательными или нет.

Мы также можем безопасно выполнить два других типа изменений: ранее существующее обязательное свойство, такое как `type`, можно превратить в необязательное, и можно слегка изменить характеристики свойства. Например, диапазон `amt` (в центах) можно изменить с 0 до 1 000 000 на более широкий: от 0 до 1 500 000. Также возможно увеличить максимальную длину запрашиваемой строки или количество элементов в массиве. Обратите внимание, что для REST API все это также применимо и к параметрам запроса, и к заголовкам HTTP-запроса.

Мы видели, что неосторожное изменение входных параметров или данных цели может привести к ошибкам, а также к критическим изменениям. С более широкой точки зрения изменение того, как API обеспечивает ответные сообщения, будь то успех или ошибка, может привести к критическим изменениям.

### *9.1.3 Как избежать критических изменений в сообщениях об успехе или ошибках*

В зависимости от используемого протокола реакция на то, как проходила обработка запроса, может быть разной, но обычно она основана на сочетании данных, возвращаемых в ответ на запрос, и некоторых функций протокола. Сначала поговорим о данных.

Для сообщений об успехе или ошибках возвращаемые данные можно безопасно изменить, основываясь на знаниях, полученных из раздела 9.1.1. Там вы увидели, как изменить сообщение об успехе, поэтому давайте попробуем изменить существующее сообщение об ошибке.

Как показано на рис. 9.6, мы могли бы изменить сообщение об ошибке цели «Перевести деньги», чтобы внести некоторые критические изменения. В модифицированной версии свойство `items` было переименовано в `errors`. Потребители не смогут получить подробную информацию об ошибке, необходимую для устранения проблемы, потому что они ожидают найти ее в items. То же самое касается значений свойства `type`: `MISSING_SOURCE` и `MISSING_DESTINATION`, которые были заменены универсальным значением `MISSING_MANDATORY`. Потребители не смогут интерпретировать новый тип ошибки.

С точки зрения чистых данных это означает, что данные ответного сообщения об успехе или ошибках должны обрабатываться одинаково, когда речь идет об их изменении, чтобы избежать критических изменений. С функциональной точки зрения второе критическое изменение означает, что мы не можем вносить новые типы ошибок в существующие цели или изменять имеющиеся.

Этот пример демонстрирует интересную вещь, касающуюся критических изменений, а говоря более конкретно, масштабов их последствий. Изменение значения `type` оказывает локальное влияние на саму цель, но переименование свойства `items` в `errors` оказывает более глобальное влияние.

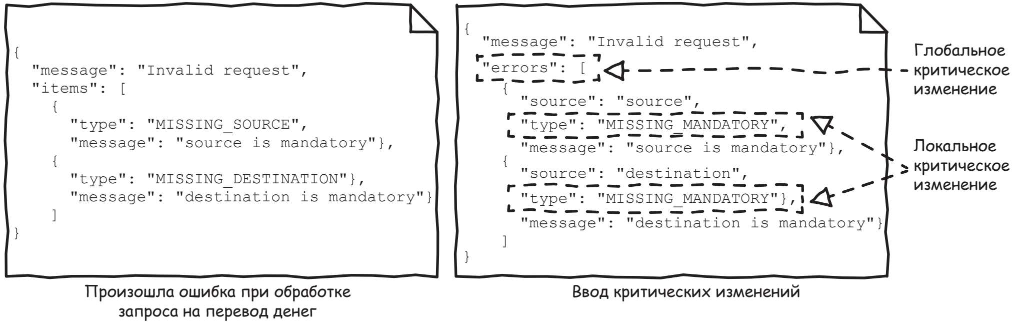



**Рис. 9.6. Критические изменения в сообщениях об ошибках**

И в самом деле, поскольку структура данных сообщений об ошибках, вероятно, одинакова во всем API, это переименование могло быть сделано не только для цели «Перевести деньги», но и для всех целей. Это означает, что ни один потребитель не сможет интерпретировать сообщение об ошибке какой-либо цели, не обновив свой код. Это критическое изменение.

Критические изменения с таким широким воздействием не ограничиваются ошибками, но они могут произойти при изменении любой общей функции API. Например, изменение соглашений об именах для идентификаторов ресурсов в соответствии с более подходящими рекомендациями – это глобальное изменение, затрагивающее входные и выходные данные. Поэтому лучше подумать дважды и применить то, что вы узнали в этой главе о возможностях избежания критических изменений. Что касается функций протокола, давайте посмотрим, что может произойти, если мы изменим коды состояния HTTP (рис. 9.7).

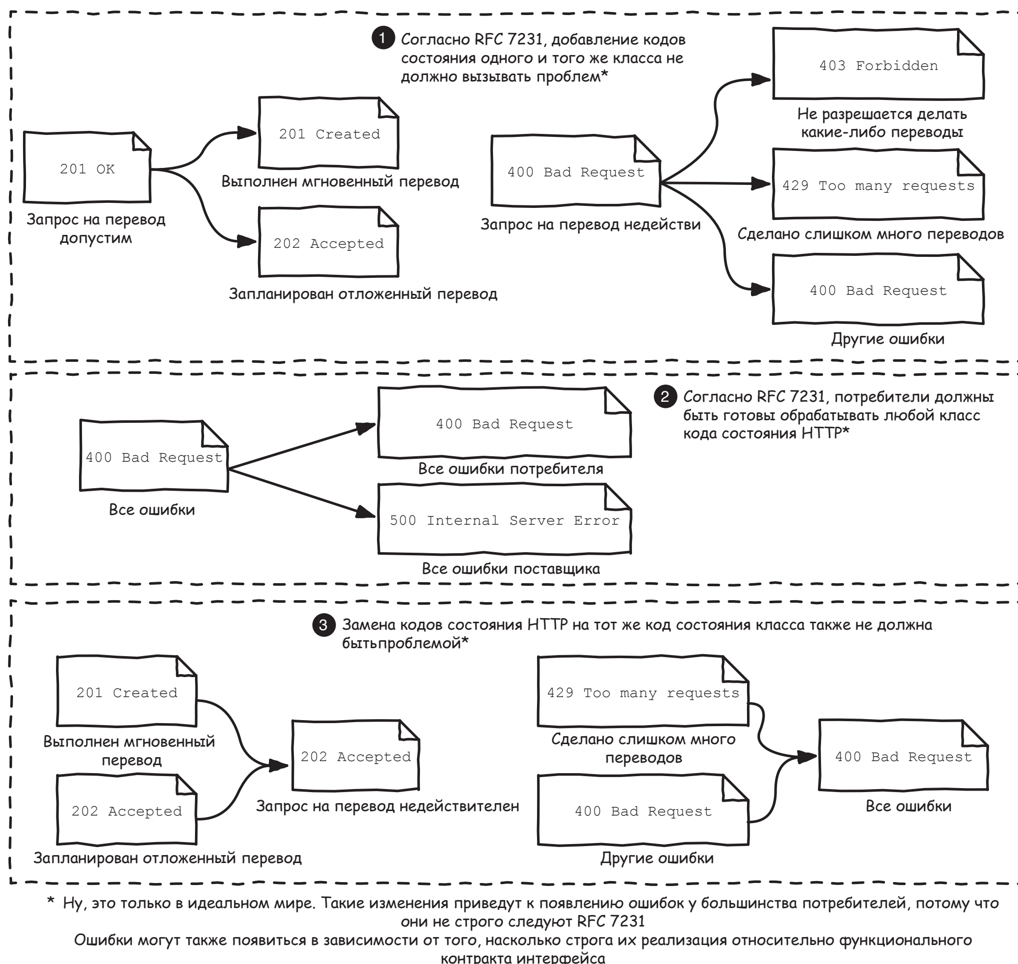



**Рис. 9.7. Изменение кодов состояния HTTP**

Спецификация RFC 7231, описывающая протокол HTTP 1.1, гласит:

> *«HTTP-клиенты не обязаны понимать значение всех зарегистрированных кодов состояния, хотя такое понимание, очевидно, желательно. Однако клиент ДОЛЖЕН понимать класс любого кода состояния, как указано в первой цифре, и воспринимать нераспознанный код состояния как эквивалентный коду состояния x00 этого класса...»*
>
> *RFC 7231*

Это означает, что добавление новых кодов состояния вообще не должно быть проблемой. HTTP-клиент должен воспринимать нераспознанный код состояния как код, эквивалентный коду статуса x00 этого класса. Таким образом, при использовании цели «Перевести деньги» потребитель, получающий неизвестную (до сих пор) ошибку `201 Created`, должен воспринимать ее так же, как и `200 OK`. То же самое происходит при

получении новой ошибки `429 Too many requests`; потребители должны воспринимать ее как базовую ошибку `400 Bad Request`.

Это также означает, что возвращение нового класса, такого как 5XX, вообще не должно быть проблемой. И в самом деле, «клиент ДОЛЖЕН понимать класс любого кода состояния». Следовательно, даже если не было известно, что цель «Перевести деньги» возвращает ошибку `500 Internal Server Error` в соответствии с документацией, потребители должны быть готовы ее обработать. А замена кодов состояния кодами одного и того же класса также не должна вызывать слишком много проблем: код ответа `201 Created` значит успех, точно так же, как и `202 Accepted`. Один и тот же код класса означает один и тот же тип ошибки. И, очевидно, что замена кода ответа `429 Too many requests` на более общий `400 Bad Request` делает ответное сообщение менее точным.

Все это хорошо, но только если мы живем в идеальном мире, а это не так. Некоторые потребители могут быть реализованы только на основании того, что написано в документации, а не в строгом соответствии со спецификацией RFC 7231. Таким образом, неожиданная ошибка `500 Internal Server Error`, вероятно, вызовет неожиданную ошибку потребителя. Некоторые потребители тоже могут слишком строго следовать вашему функциональному интерфейсу контракта. Если отложенный денежный перевод подтвержден кодом состояния `201 Created`, при возвращении кода `202 Accepted`, вероятно, в некоторых из них появятся ошибки, даже если вы предоставите общее и информативное ответное сообщение, потому что потребители ожидают только надпись `201 Created`, и ничего больше. Таким образом, вы должны надеяться, что потребители неукоснительно соблюдают спецификацию RFC 7231 и не слишком строго следуют вашему функциональному контракту интерфейса. Это довольно сложно, если вы не работаете в тесном контакте с ними.

Единственное некритическое изменение, которое можно было бы сделать ничего не опасаясь, – это удалить код состояния HTTP, потому что основная ошибка никогда не произойдет из-за изменений в реализации. Все другие модификации, даже если они должны быть приняты в соответствии со спецификацией RFC 7231, должны выполняться крайней осторожно. Никогда не доверяйте потребителям, если вы не знаете, как они на самом деле написаны. Конечно, когда речь идет о протоколе HTTP. Если ваш API использует другой протокол, вам нужно будет проверить, как этот протокол работает, чтобы определить, как лучше всего обрабатывать изменения в ответных сообщениях, основываясь на том, что вы здесь узнали.

### 9.1.4 Как избежать критических изменений в целях и потоках

Критические изменения также могут происходить на более высоком уровне при изменении целей и потоков. Что касается целей, мы уже знаем, что изменение входных и выходных данных или ответных сообщений может привести к серьезным изменениям, но это еще не все.

Есть два других очевидных способа внесения критических изменений: переименование или удаление целей. Например, банковский API пред-

лагает цели «Перевести деньги» и «Перечислить переводы». Эти цели обозначаются с помощью запросов `POST /transfers` и `GET /transfers` соответственно. Если мы решили переименовать ресурс перевода в `money-transfers`, потребители, использующие эти две цели, получат в ответ ошибку `404 Not Found`. Можно было бы использовать код состояния `301 Moved Permanently` для перенаправления всех вызовов для `/transfers` в `/money-transfers`. Но это работает, только если потребители понимают и фактически следуют перенаправлению, как показано в приведенном ниже листинге.

**Листинг 9.1. Активация флага redirects в Java HttpUrlConnection**

```
URL obj = new URL(«https://api.bankingcompany.com/transfers»);
HttpURLConnection conn = (HttpURLConnection) obj.
openConnection();
conn.setInstanceFollowRedirects(true);
HttpURLConnection.setFollowRedirects(true); ①
```

① **Перенаправление не будет выполнено без явной настройки флага.**

Такая конфигурация может быть просто неизвестна (не все являются экспертами по протоколу HTTP), и она также может быть намеренно деактивирована по соображениям безопасности. Некоторые потребители, возможно, не захотят отправлять свои запросы куда-либо без их одобрения и, скорее всего, предпочтут получить ошибку в своем коде.

Другим очевидным критическим изменением будет удаление цели. Если мы решим удалить метод `GET` для ресурса перевода, потребители, использующие цель «Перечислить переводы», получат в ответ ошибку `405 Method Not Allowed`. Очевидно, что лучше не удалять и не переименовывать цели, но означает ли это, что можно добавлять цели по своему усмотрению?

Предположим, что по соображениям безопасности проектировщики банковского API решили, что каждый денежный перевод должен проверяться владельцем исходного счета с использованием одноразового пароля (OTP), получаемого посредством SMS-сообщения. Один из способов справиться с этой модификацией – добавить новую цель «Проверить перевод», которая должна вызываться после перевода денег. Она может ожидать идентификатора перевода и этот одноразовый пароль, отправляемый после получения запроса на перевод денег. Интерфейс цели «Перевести деньги» не изменяется вообще, но, поскольку необновленные потребители не будут вызывать новую цель «Проверить перевод», они больше не смогут инициировать перевод денег. И даже хуже – поскольку ошибки нет, они не будут знать о проблеме; это – *безмолвное* критическое изменение.

Добавление новой цели «Безопасно перевести деньги» ничего не нарушит, но и не обеспечит никакой защиты, потому что потребители по-прежнему смогут вызвать первоначальную цель для небезопасного перевода денег. Ввод нового обязательного шага в существующий поток является критическим изменением, как и изменение поведения су-

ществующих целей. Все, что можно сделать на уровне цели или потока, – добавить совершенно новые цели, которые потребители не должны использовать для существующих потоков. Но при этом вы должны обратить внимание на безопасность.

### 9.1.5 *Предотвращение нарушений в системе безопасности и критических изменений*

Модификация API может привести к критическим изменениям, которые влияют на безопасность и приводят к риску нарушений в системе безопасности; поэтому все модификации API должны выполняться с учетом безопасности. По сути, вы должны применять все, что узнали в главе 8, при модифицировании API каким-бы то ни было способом. Например, когда речь идет о данных, добавляемых к ответам существующих целей, необходимо убедиться, что эти данные не будут предоставлены потребителям, которые не должны их получать.

Вы также должны быть осторожны при изменении групп. Некоторые изменения могут привести к нарушениям в системе безопасности или критическим изменениям, как показано на рис. 9.8.

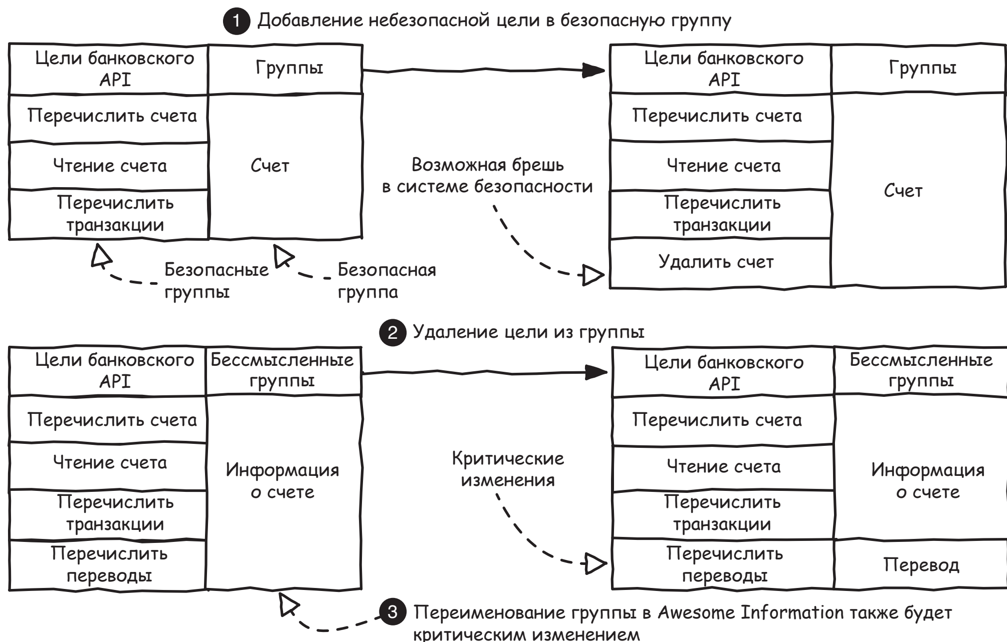



**Рис. 9.8.** Нарушения в системе безопасности и критические изменения при изменении групп

Во-первых, в зависимости от выбранной стратегии разбиения API (см. раздел 8.2) введение цели «Удалить счет», представленной, например, запросом `DELETE /accounts/{accountId}`, может быть проблематичным. Если разбиение основано на ресурсах, любой потребитель, имеющий доступ к ресурсу `/accounts`, получит доступ к этой вполне разумной, но опасной цели.

Во-вторых, если добавление новой цели в существующую группу требует осторожности, как насчет того, чтобы удалить ее? Допустим, группа доступа к информации о счетах включает в себя цели «Перечислить счета», «Прочитать счет», «Перечислить транзакции» и «Перечислить переводы». Основываясь на том, что вы прочитали в разделе 8.2, мы знаем, что такое разбиение, охватывающее различные темы, довольно неудобно.

Возможно, нам следует убрать из этой группы цель «Перечислить транзакции», чтобы сделать ее более понятной. Но это будет означать, что существующие потребители с группой доступа к информации о счетах больше не смогут ее использовать. Таким образом, удаление цели из группы вводит критическое изменение. И наконец, переименование или удаление группы будет иметь тот же эффект: потребители потеряют доступ ко всем целям, которые были ей охвачены.

Обычно проектировщики API не должны решать, как API фактически защищен, но вы должны знать, что изменение аспектов безопасности API может привести к критическим изменениям. Прежде чем продолжить читать, подумайте о том, что вы узнали в главе 8, и постарайтесь выяснить, каким образом изменения безопасности могут привести к критическим изменениям или нарушениям в системе безопасности. Я вернусь через минуту. Когда вы закончите, можете прочитать то, что написано далее.

Исходя из того, что вы уже узнали, вы должны понимать, что изменение способа получения токенов (например, замена потока OAuth 2.0 на другой) непоправимо приведет к критическим изменениям на стороне потребителя. Изменение способа их передачи в запросе (изменение данных) также серьезно. И последнее, но не маловажное: поскольку идентификация обработки приложения/системы может быть независимой от API, его можно изменять без ведома людей, отвечающих за реализацию API. Такое изменение данных безопасности, прикрепленных к токенам, может иметь ужасные последствия для реализации.

Например, удаление идентификатора конечного пользователя из данных, прикрепленных к токену доступа, в лучшем случае приведет к неожиданным ошибкам сервера, а в худшем – к нарушениям в системе безопасности; реализация будет думать, что, поскольку конечный пользователь не участвует, потребитель относится к типу администратора. Всегда полезно знать об этом и напоминать другим людям, работающим над API, о влиянии таких модификаций.

### 9.1.6 Невидимый контракт интерфейса

Пока что, то, что вы видели, касается видимой части контракта интерфейса: все, что может быть описано с использованием формата описания API или документации. Но некоторые потребители также могут использовать и *невидимые* части контракта API.

Например, у владельца счета могут быть разные адреса. Эти адреса возвращаются в списке, и у каждого есть свойство `type`, указывающее на то, домашний ли это адрес, адрес офиса или временный. Допустим, не-

которые проницательные проектировщики заметили, что адреса всегда идут в таком порядке: `home`, затем `office`, а потом `temporary`. Поэтому, когда они хотят получить домашний адрес, они используют его индекс (0), вместо того чтобы сканировать список в поисках адреса с типом `home`. Мы все согласны с тем, что это полная чушь; потребители не должны этого делать. Но если они это сделают и если этот порядок изменится, эти потребители укажут неправильный адрес.

Еще один пример этого невидимого контракта интерфейса – потребители могут решить, что длина метки транзакции, которая просто описывается как строка без каких-либо других подробностей, не может превышать 50 символов, основываясь на данных, которые они уже получили. Мы уже знаем, что может произойти, если длина этих меток будет увеличена: возможны ошибки базы данных. Как видите, потребители могут полагаться на фрагменты API, которые явно не описаны. Закон Хайрама гласит:

> *«При достаточном количестве пользователей API не имеет значения, что вы обещаете в контракте: все наблюдаемые действия вашей системы будут зависеть от кого-то».*
>
> *Закон Хайрама (Хайрам Райт)*

Что произойдет, если изменить цель «Перевести деньги» с чисто внутренней точки зрения (без внесения каких-либо изменений в видимый контракт интерфейса), чтобы добавить новые элементы управления, которые немного увеличивают время отклика цели? Работа некоторых потребителей, настроивших свои тайм-ауты в соответствии с фактическим временем отклика, может быть нарушена, потому что новая версия цели занимает больше времени, чем их значение тайм-аута. Эти соображения могут быть неочевидны, но любой проектировщик API (или любой, кто работает над API) должен знать невидимые части контракта интерфейса для правильной оценки важности любых изменений, внесенных в API.

Мы рассмотрели множество разных способов введения страшных критических изменений. Но всегда ли стоит их бояться?

### 9.1.7 Критическое изменение – не всегда проблема

*Критическое изменение* – это изменение, которое вызовет проблемы у потребителей, если они не обновят свой код. Как вы уже видели, эти проблемы также могут иметь последствия на стороне поставщика. Если потребители банковского API – это сторонние приложения, разработанные другими компаниями, внесение критических изменений в API определенно не вариант. Работа потребителей будет нарушена, а их разработчики расстроятся, потеряют доверие к банковскому API и, возможно, решат использовать вместо него конкурирующий API. Следовательно, банковская компания может потерять деньги – если не хуже.

Но не все API являются открытыми и используются тысячами сторонних потребителей. Если бы банковский API был закрытым и действовал в качестве простого серверной части для одностраничного приложения,

а также мобильного приложения, созданного самой банковской компанией, критические изменения были бы практически осуществимы. Все, что требуется в этом случае, – обновить одностраничное приложение на веб-сервере банковской компании, где размещены его файлы, и принудительно выполнить обновление мобильного приложения, при условии, что это приложение включает в себя функцию принудительного обновления.

Как видите, в зависимости от контекста внесение критических изменений может и не быть проблемой, если все потребители могут обновляться синхронно с API. Но, честно говоря, это может быть непростой задачей. Наиболее безопасный вариант, если критические изменения неизбежны, – это управление версиями API.

## 9.2 Управление версиями API

Вот и настал этот день! Банковская компания решила запустить версию № 2 своего знаменитого банковского API. На рис. 9.9 показан возможный сценарий среди множества других для обработки такого изменения.

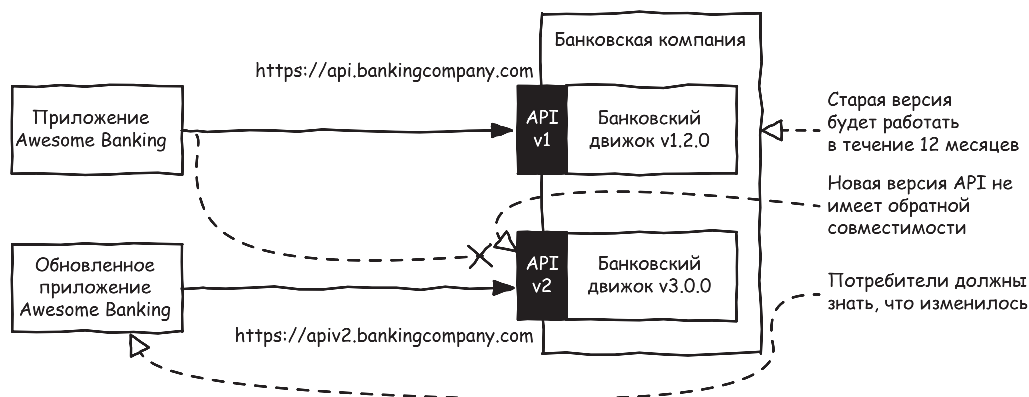



**Рис. 9.9.** Банковская компания обновила свой банковский API до версии 2

Новая версия банковского API доступна по адресу apiv2.bankingcompany.com. Потребители, переходящие на новую версию, получают удивительные функции, такие как возможность осуществлять международные денежные переводы в любой валюте благодаря новому банковскому движку 3.0.0, написанному на языке Go, который пришел на смену движку 1.2.0, написанному на старом добром COBOL. Но к сожалению, он не полностью обратно совместим.

При переходе на новую версию потребителям также понадобится обновить свой код, поскольку некоторые цели, такие как «Перечислить транзакции», были изменены без учета обратной совместимости, чтобы соответствовать новым функциям. Банковская компания также объявила, что будет поддерживать версию 1 (которая представлена на сайте api.bankingcompany.com) в течение 12 месяцев. Это означает, что у потребителей есть 12 месяцев для обновления, даже если они не используют из-

мененные функции или намереваются использовать новые. Это также означает, что банковской компании придется запустить две версии серверной части на 12 месяцев.

В этом случае, с точки зрения проектировщика API, работа с версиями заключается в проектировании без учета обратной совместимости – внесении критических изменений и изменении имени домена для различия двух версий API. Но управление версиями API – это тема, выходящая за рамки простого проектирования API, и проектировщики API, как и любой другой человек, работающий над API, должны знать обо всех его последствиях.

Помимо проектирования, управление версиями API оказывает влияние на реализацию и управление продуктом. Выбор стратегии управления версиями влияет не только на то, как вы проектируете API, но и на то, как вы его реализуете (банковская компания предоставляет две версии API, используя две отдельные серверные части). Кроме того, то, что вы предоставляете новую версию своего продукта (своего API), не означает, что потребители захотят перейти на него; многие, возможно, предпочитают придерживаться предыдущей версии. Однако, прежде чем исследовать различные способы представления версий API и их влияние, давайте выясним, что такое управление версиями API и в чем состоит его отличие от управления версиями реализации.

### 9.2.1 Управление версиями API и реализации

Первоначальная версия банковского API предоставляла доступ только к информации о счете, но быстро развивалась, предлагая больше возможностей. На рис. 9.10 показана эволюция этого API и его реализация.

Сразу после запуска API был обновлен, чтобы обеспечить возможность совершения денежных переводов. Очевидно, нужно обновить реализацию, чтобы обеспечить новые цели, связанные с переводами. После этого обновления API и реализация использовали одну и ту же версию с номером 1.1. К сожалению, первая версия реализации переводов не была по-настоящему эффективной. Каждый денежный перевод обрабатывался синхронно при каждом вызове API. Это приводило к длительному времени отклика, особенно когда было более 100 запросов на перевод в секунду. Затем было решено изменить реализацию, чтобы помещать запросы на перевод денег в очередь сообщений и обрабатывать их асинхронно, не влияя на API.

Новая реализация 1.2 была намного более эффективной, но по-прежнему предоставляла доступ к тому же API версии 1.1. После этого технический директор банковской компании стал поклонником языка Go и решил избавиться от COBOL. Первой попыткой было автоматически преобразовать код COBOL в код Go. Хотя в банковском API использовался совершенно другой язык программирования, реализация версии 2.0 смогла представить доступ к версии 1.1, поэтому потребители вообще не заметили этого изменения. К сожалению, до запуска в производство было выявлено, что сгенерированный код неэффективен и плохо напи-

сан. Таким образом, код был полностью переписан вручную, а также добавились долгожданные новые функции. Но самым важным изменением явилось то, что самые старые

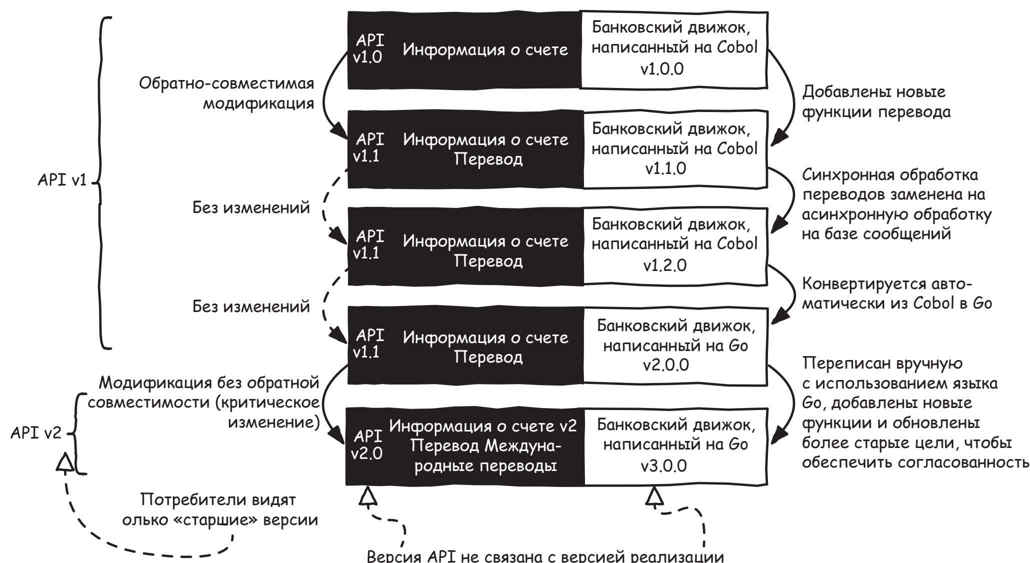



**Рис. 9.10.** Эволюция банковского API и его реализация

цели API, связанные с информацией о счетах, были изменены, чтобы соответствовать правилам проектирования API, введенным с функциями переводов в версии 1.1. Это критическое изменение заставило банковскую компанию обновить API до версии 2.0, не совместимой с предыдущими версиями.

Как видите, у API есть версия, как и у любого программного компонента, но она не связана с версией его реализации. Версия API развивается на основе изменений, внесенных в контракт интерфейса (изменений, которые видны с точки зрения потребителей), а не того, как развивается реализация. Две совершенно разные реализации могут предоставлять одну и ту же версию API. В этом примере имена API и версий реализации, такие как v1.1 или v3.0.0, основаны на хорошо известной и умной системе использования чисел для именования версии компонента программного обеспечения – *семантического управления версиями* (https://semver.org/). Она заключается в использовании трех цифр в формате: *MAJOR.MINOR.PATCH*. Каждая цифра увеличивается в определенных ситуациях:

- первая цифра – старшая версия (MAJOR) – увеличивается только при критических изменениях, таких как добавление нового обязательного параметра (см. раздел 9.1);
- вторая цифра – младшая версия (MINOR) – увеличивается при добавлении новых функций обратно-совместимым способом, например при добавлении новых методов HTTP или путей к ресурсам в REST API;

- третья цифра – номер патча (PATCH) – увеличивается, если внесенные изменения включают в себя исправления ошибок, совместимые с предыдущими версиями.

Это имеет смысл для реализации, но не для API. Семантическое управление версиями, применяемое к API, состоит из двух цифр: *BREAKING.NONBREAKING*. Это двухуровневое управление версиями интересно с точки зрения поставщика; оно помогает отслеживать различные обратно совместимые и несовместимые версии API. Но потребителей не интересуют все эти детали.

Потребители, которые использовали цели, связанные с информацией о счетах версии API 1.0, могут легко перейти на версию 1.1 (*NONBREAKING*), как будто ничего и не изменилось. И даже если они решат использовать новые функции перевода, добавленные в версию 1.1, они просто используют банковский API, не заботясь (или не зная) о точном номере версии.

В действительности потребители заметят изменения в API только тогда, когда банковская компания представит версию 2.0. Фактически им придется изменить некоторые части своего кода, чтобы использовать его. С точки зрения потребителей API, они просто используют версию 1 или версию 2. Их не волнует второй уровень управления версиями (*NONBREAKING*); им нужна только цифра BREAKING.

> **ПРИМЕЧАНИЕ.** Помните, что критическое изменение не является обратно совместимым. Это может быть очевидная модификация контракта интерфейса или более коварная модификация невидимого контракта.

Если удаление или переименование цели приводит к значительному увеличению версии, возможно, не стоит делать то же самое для невидимой модификации, которую мы обсуждали в разделе 9.1.6. Это нужно обсуждать отдельно для каждого индивидуального случая, и нужно оценить его реальное воздействие на потребителей, чтобы определить, необходим ли в таких случаях выпуск новой версии.

Если для потребителей важен только один уровень управления версиями, можно использовать в качестве имен версий все что угодно. Мы могли бы использовать даты в формате ISO 8601, такие как 2017-10-19 для версии 1 и 2018-22-12 для версии 2. Если бы мы хотели, то могли бы даже использовать имена известных композиторов саундтреков к аниме, таких как Йоко Канно и Кэндзи Кавай для версий 1 и 2 соответственно.

Управление версиями API и реализации различаются, и потребителей (в основном) волнуют только изменения версий, объявляющих о критических изменениях. Но как потребители сообщают, какую версию API они хотят использовать?

### 9.2.2 Выбор представления управления версиями API с точки зрения потребителя

Банковская компания выпустила совершенно новый банковский API версии 2.0, который не полностью обратно совместим. Надеемся, что

цели, связанные с переводами, обратно-совместимы, поэтому пользователям, использующим цели, которые были в предыдущей версии API, нужно будет лишь немного подправить свои запросы, чтобы перейти на новую версию. На рис. 9.11 показаны разные возможности, которые может выбрать банковская компания, чтобы предоставить доступ к различным версиям API.

Банковский API может использовать путь к ресурсу для обработки версии API. Потребители, желающие получить список переводов, могут отправить запрос `GET /v1/transfers` или `GET /v2/transfers` на тот же домен api.bankingcompany.com, чтобы использовать версию API 1 или 2 соответственно. Аналогичным подходом будет использование разных доменов или поддоменов для каждой версии API: в данном случае api.bankingcompany.com для версии 1 и apiv2.bankingcompany.com для версии 2.

Версию используемого API также можно указать с помощью параметра запроса (`GET /transfers?version=2`) или пользовательского заголовка (`Version: 2`). Или банковский API может предложить указывать желаемую версию API с помощью согласования содержимого, о котором вы узнали из раздела 6.2.1.

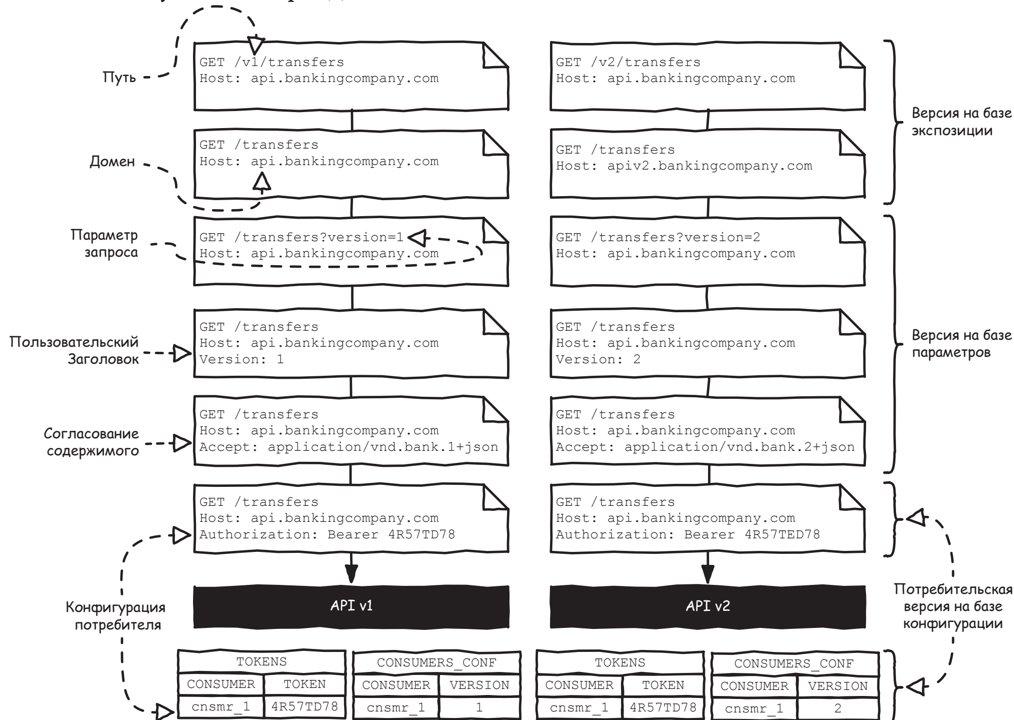



**Рис. 9.11. Различные способы указания версии API в запросе**

Для этого потребители указывают собственный медиатип в стандартном заголовке `Content-type`, например `application/vnd.bank.2`, чтобы сообщить, что они хотят использовать API версии 2.

И наконец, что не менее важно, версия используемого API может быть косвенно указана в запросе. Поскольку API является безопасным, потребители должны отправлять с каждым запросом некие учетные данные; при таком подходе запрос содержит заголовок `Authorization` с токеном (в данном случае это `4R57TD78`). Согласно данным, хранящимся у поставщика в таблице `TOKENS`, этот токен был сгенерирован для потребителя `cnsmr_1` (очевидно, в реальном мире никто никогда не будет хранить такие конфиденциальные данные без шифрования). Версия API, используемая этим потребителем, указана в столбце `VERSION` в таблице `CONSUMERS_CONF`.

Вот шесть различных способов указания версии для API на базе HTTP. Какой из них выбрать? Очевидно, этот выбор должен быть сделан с точки зрения потребителя.

Самый простые варианты – управление версиями пути и домена. Изменить доменное имя или путь в URL-адресе довольно просто, особенно если API тестируется с помощью браузера или утилиты командной строки `curl`. Потребители могут увидеть, какая версия применяется, посмотрев на URL-адрес, который они используют. Это, вероятно, наиболее используемые варианты; и, основываясь на том, что вы узнали из раздела 6.1.4, это стоит принять во внимание. Ваши потребители, которые, вероятно, уже знакомы с этими механизмами, найдут их простыми в использовании.

Управление версиями параметров запроса также является довольно простым вариантом; но, с точки зрения проектировщика API, я не рекомендую его, потому что он не совсем безупречен. Например, если мы добавим фильтр валют, как в `GET /transfers?currency=eur&version=2`, запрос будет смешивать чисто технический параметр с функциональным.

Управление версиями `Content-type` интересно, с точки зрения эксперта по HTTP, но многие люди не хотят использовать заголовки HTTP, несмотря на то что это совсем не сложно. Эта проблема усугубляется пользовательским заголовком HTTP, потому что это не является частью стандарта HTTP.

Вариант с конфигурацией потребителя полностью удобен для потребителя, поскольку потребителям не нужно изменять свой код. Один небольшой недостаток заключается в том, что для перехода с одной версии на другую требуется обновление конфигурации, что может быть неудобно при тестировании разных версий API.

Что бы выбрали вы? Лично я предпочитаю управление версиями пути и конфигурации потребителя, но давайте вернемся назад и рассмотрим не только REST, HTTP и личные предпочтения.

Мы видим, что существует три способа предоставления доступа к разным версиям API. Первый – просто рассматривать новую версию как новый API и создать новую конечную точку. Второй – сохранить единую конечную точку для различных версий, но передавать параметр в запросах, применяя функции протокола или метаданные в данных запроса, которые указывают на версию используемого API. В третьем способе также берется единая конечная точка, но версия, используемая каждым

потребителем, хранится на стороне поставщика. Независимо от принятого технического решения выбор способа указания версии этого API должен учитывать стандарты и удобство использования, чтобы потребители могли понять его и с легкостью применять.

До сих пор мы говорили об управлении версиями целого API. Но является ли это единственным вариантом?

### 9.2.3 Выбор детализации

Управление версиями API в целом является наиболее распространенной практикой, но не единственной. В зависимости от варианта использования и типа API другие варианты могут быть более эффективными.

В случае с REST API, помимо уровня API, управление версиями может выполняться на уровне ресурсов, уровне целей и операций и уровне данных и сообщений. На рис. 9.12 приведено сравнение управления версиями API и версиями ресурсов при внесении критических изменений. Обратите внимание, что критических изменений в этом примере можно избежать, основываясь на том, что вы узнали из раздела 9.1.

Чтобы упростить пример, банковский API сводится к трем целям: перевод денег, перечисление переводов и удаление перевода. В левой части показано, что происходит при управлении версиями API как единого целого, а в правой – что происходит при управлении версиями каждого ресурса, определенного его путем. В обеих частях номер версии находится на первом уровне пути. Посмотрите на верхнюю часть рисунка. Слева – API версии 1. Справа – версия двух ресурсов перевода (`/v1/transfers` и `/v1/transfers/{ID}`) – версия 1.

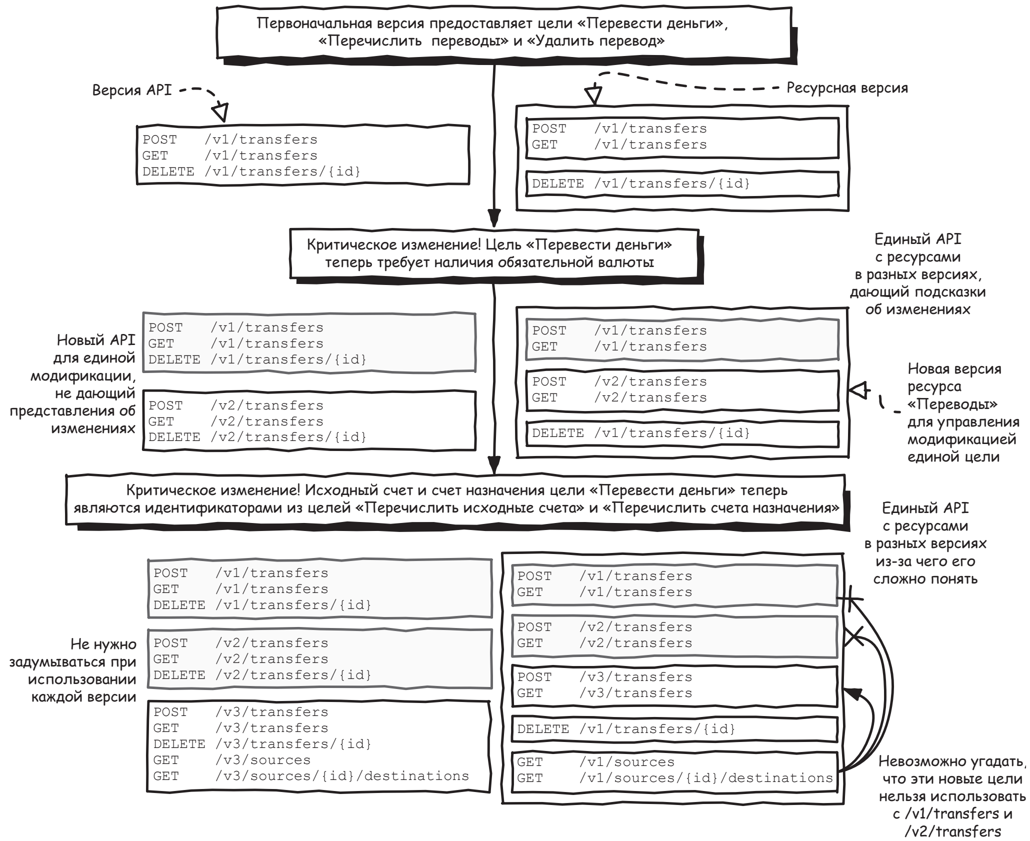



**Рис. 9.12. Управление версиями API в сравнении с управлением версиями ресурсов**

Цель «Перевести деньги» предполагает наличие номера исходного счета (`source`) и счета, куда будут переведены средства (`destination`), а также сумму денег (`amount`). Первое критическое изменение вводится путем добавления нового обязательного свойства `currency` в исходные данные этой цели. На стороне управления версиями API это единственное критическое изменение вынуждает нас создавать новую версию API – v2. Если потребители сравнят цели двух версий, они не будут иметь абсолютно никакого представления о том, что изменилось, не прочитав замечания к версии API.

Что касается управления версиями ресурса, новый API не создается, но добавляется новый ресурс `/v2/transfers` для управления модификацией операции `POST /v2/transfers`. Это дает подсказку потребителям, но невозможно узнать, какая операция с ресурсом перевода была изменена, не читая замечания к версии.

**ПРИМЕЧАНИЕ.** Замечания к версии программного продукта всегда должны присутствовать, но они не всегда доступны при необходимости, а некоторые люди вообще их не читают! Возможность обна-

ружения изменений может быть полезна для потребителей или людей, вовлеченных в проект.

Также были введены две новые цели, и цель «Перевести деньги» была изменена, чтобы облегчить перевод денег и управлять переводами на внешние счета. Цель «Перечислить источники» позволяет потребителю перечислить все возможные источники для перевода денег, а цель «Перечислить счета назначения» дает список счетов, куда должны уйти деньги. Ввод этих двух новых целей не является критическим изменением. Но, к сожалению, исходные счета и счета назначения идентифицируются номером, отличающимся от строковых номеров счетов, которые ожидает увидеть цель «Перевести деньги». Ее входные данные изменяются, тем самым внося критические изменения.

На стороне управления версиями ресурса добавлен новый ресурс перевода версии 3 с путем `/v3/transfers` наряду с операциями `GET /v1/sources` и `GET /v1/sources/{id}/destination`. API теперь включает в себя три разные версии ресурса перевода, а новые ресурсы исходного счета и счета назначения могут использоваться только с версией 3. Потребителям будет не так-то просто догадаться об этом.

Что касается управления версиями API, снова создается новый API версии 3, но нет необходимости думать о том, какие версии можно использовать вместе. Каждая независимая версия API содержит набор совместимых ресурсов.

Давайте теперь перейдем к более глубокому уровню управления версиями – на уровне цели или операции. На рис. 9.13 сравнивается управление версиями API и управление версиями на уровне целей и операций, когда вносятся те же самые критические изменения.

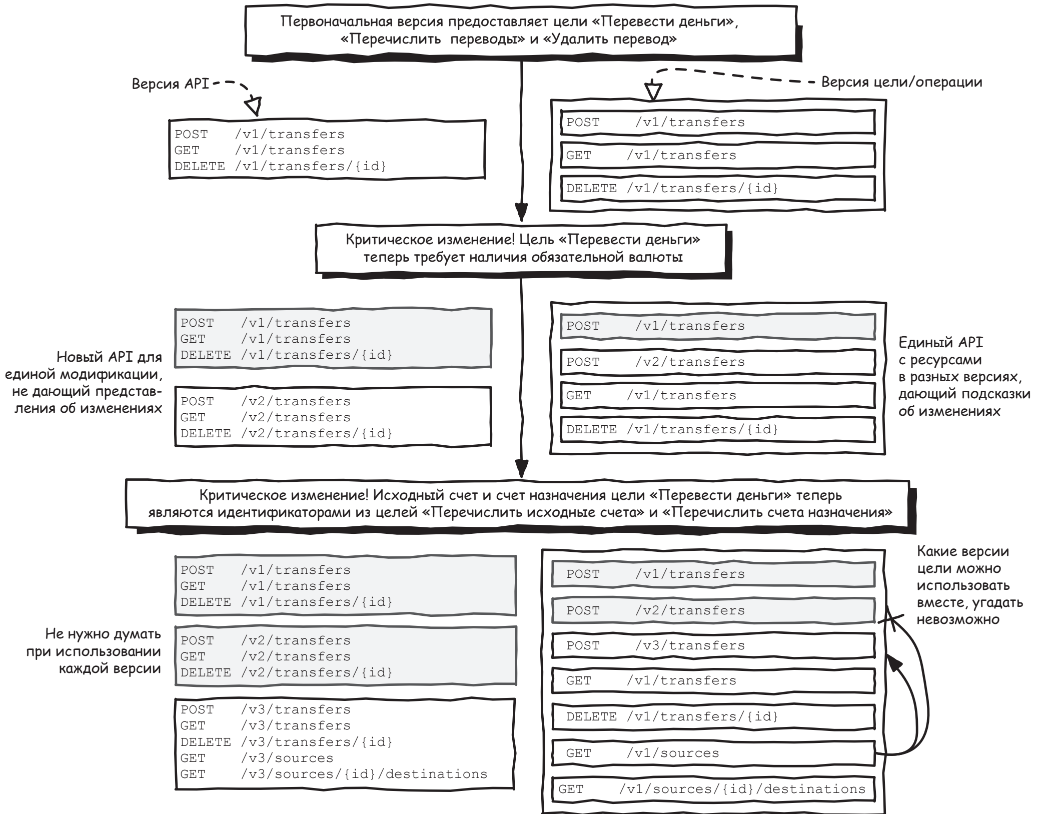



**Рис. 9.13. Управление версиями API и управление версиями на уровне целей (или операций)**

Управление версиями API точно такое же; но, с другой стороны, каждая операция теперь версионируется независимо, по-прежнему используя vX в пути. При каждом критическом изменении цели «Перевести деньги» в API добавляется новый запрос (`POST /v2/transfers`, затем `POST /v3/transfers`). Это полезно, потому что ясно показывает, какая цель была изменена. Но, как и в случае управления версиями ресурсов, у API получается три разных версии цели «Перевести деньги», и потребители не имеют ни малейшего понятия, что запросы `GET /v1/sources` и `GET /v1/sources/{id}/destinations` могут использоваться только с `POST /v3/transfers`, что определенно не подходит потребителям.

Давайте теперь посмотрим на последний уровень контроля версий: уровень данных и сообщений. На рис. 9.14 показано, что может произойти с целью «Перевести деньги» при использовании этой стратегии. Обратите внимание, что эта стратегия работает только с данными, расположенными в теле запросов и ответов; заголовки и параметры запроса сюда не входят.

**Рис. 9.14.** Управление версиями данных (или сообщений) с использованием согласования содержимого

Поскольку в банковском API используется протокол HTTP, мы можем применить функцию согласования содержимого, чтобы управлять версиями запросов и ответов для каждой цели. В первоначальной версии API запросы `POST /transfers` и ответы на него используют специальный медиатип `application/vnd.transfer.request.v1+json`. Когда вводные данные цели «Перевести деньги» меняются, версия ее медиатипа повышается до цифры 2 (`application/vnd.transfer.request.v2+json`), а затем до 3 (`application/vnd.transfer.request.v3+json`). Версия ответа изменяется только при втором критическом изменении до версии 2.

В обоих случаях версии запросов и ответов больше не соотносятся друг с другом, и при втором обновлении становится неясно, какие версии запросов и ответов работают вместе. Обратите внимание, что мы могли бы идеально сопоставить версии сообщений с запросами и ответами, просто объединяя версии запросов и ответов независимо от того, какая из них была изменена. В этом случае такая стратегия близка к стратегии управления версиями на уровне целей и операций.

В табл. 9.3 собраны положительные и отрицательные стороны каждого уровня детализации.

**Таблица 9.3. Выбор степени детализации управления версиями для REST API**

| Детализация | Плюсы | Минусы | Рекомендации к использованию |
|---|---|---|---|
| API | Не нужно думать о том, какие версии операций или ресурсы работают вместе | Изменение версии API при однократном критическом изменении без понятия об изменениях | По умолчанию для REST API (обычная практика) |
| Ресурс | Дает подсказки об изменениях | Невозможно угадать, какие версии работают вместе | Не рекомендуется для REST API; использовать, только когда ресурсы полностью независимы |
| Цель/ операция | Указывает, какие цели изменились | Невозможно угадать, какие версии работают вместе | Не рекомендуется для REST API; использовать, только когда операции полностью независимы |
| Данные/ сообщение | Указывает, какие данные/ сообщения изменились | Невозможно угадать, какие версии работают вместе; ограничено телом запросов и ответов при использовании HTTP | Не рекомендуется для REST API; может использоваться в сочетании с детализацией на уровне API |

Каждый уровень детализации имеет свои плюсы и минусы, но, по крайней мере, в мире REST API наиболее часто используемая стратегия – это управление версиями на уровне API. К выбору любой другой детализации не следует подходить легкомысленно, потому что большинство потребителей не привыкли к таким стратегиям управления версиями, но это не означает, что они никогда не должны использоваться, особенно если проектируемый вами API не REST API.

Иногда вам может потребоваться смешивать разные уровни детализации. Например, если вы работаете в банковской сфере, вам, возможно, придется работать со стандартом ISO 20022, который определяет сообщения в формате XML (и вскоре, на момент написания этой книги, и JSON). Сообщения приходят в версионированных парах типа запрос/ответ. Если бы вы проектировали API с использованием этих сообщений, вам пришлось бы иметь дело с управлением версиями и вашего API и сообщений ISO 20022.

Управление версиями API должно содержать несколько секретов для вас. Но, как проектировщик API, вы должны знать о его влиянии за пределами проектирования.

### 9.2.4 Влияние управления версиями API за пределами проектирования

То, что обсуждается здесь, в основном касается менеджеров по продуктам API, технических руководителей проекта и архитекторов; но, будучи проектировщиками API, вам также полезно быть в курсе этих вопросов (кроме того, иногда проектировщики API выполняют несколько ролей). Даже если изменения, внесенные в API, не являются критическими, каждое из них должно быть тщательно записано, чтобы вы могли передать список изменений потребителям.

К настоящему времени вы должны понимать, что изменение версии API – или, точнее, внесение критических изменений – имеет последствия для потребителя и потребители могут быть не довольны этим. Создание новых версий API означает, что несколько версий API будет работать одновременно, и потребители, возможно, будут не готовы приложить усилия, чтобы перейти на более новую версию, если старая версия, которую они используют, по-прежнему работает. Поэтому, критические изменения, вводимые в API, должны быть тщательно отобраны.

Например, внесение критических изменений, которые не приносят никакой пользы потребителям, такие как переход с версии фреймворка для авторизации OAuth 1 на версию 2, – определенно плохая идея. Чтобы сделать переход более удобным, было бы лучше представить новые функции, которые нужны потребителям, наряду с такими скучными критическими изменениями.

Что касается реализации, то для предоставления доступа к нескольким версиям API может потребоваться дополнительная работа и, следовательно, важно решить, сколько версий будет поддерживаться и в течение какого времени. Это зависит от вашего контекста. Некоторые компании, предоставляющие свои сервисы только в виде API, могут принять решение неограниченно поддерживать все версии. С другой стороны, в случае с закрытым API некоторые компании могут поддерживать только две версии. Универсального средства не существует; вам выбирать адаптированное решение.

На техническом уровне для управления версионированием в реализации есть два варианта. Первый вариант – каждая версия должна обрабатываться определенной реализацией. Это означает, что проектирование более старой версии реализации будет продолжено (по крайней мере, для исправления ошибок и проблем безопасности, например), пока эти версии используются. Инфраструктура, поддерживающая более старую версию, также должна обслуживаться. В зависимости от контекста вашей компании это может быть или не быть проблемой. Такой контекст определенно повлияет на то, как долго вы разрешаете потребителям использовать более старые версии API. Второй вариант – все версии обрабатываются одной реализацией. Опять же, в зависимости от контекста, когда одна реализация управляет всеми возможными версиями ваших API, это может быть, а может и не быть проблемой.

Как видите, управление версиями может быть сложным. Существуют ли способы уменьшить риск критических изменений, которые требуют изменения версии API?

## 9.3 Проектирование API с учетом расширяемости

Мы знаем, как избежать внесения критических изменений, когда это возможно, и как управлять версиями API, когда такие изменения неизбежны. Это хорошо, но мы не должны забывать об одном из фундаментальных принципов разработки программного обеспечения: расширяемости.

> *«Расширяемость – это принцип разработки программного обеспечения и системного проектирования, при котором реализация учитывает будущий рост. Термин «расширяемость» также можно рассматривать как системную меру способности расширять систему и уровень усилий, необходимых для реализации расширения. Расширения могут осуществляться через добавление новой функциональности или путем модификации существующей функциональности. Основная тема заключается в обеспечении изменений – как правило, улучшений – при минимальном воздействии на существующие функции системы».*
>
> *«Википедия»*

Тщательно проектируя данные, взаимодействия и потоки и выбирая соответствующий уровень детализации для управления версиями, мы можем проектировать расширяемые API-интерфейсы, которые способствуют изменениям, и, что более важно, снижают риск ввода критических изменений.

### 9.3.1 Проектирование расширяемых данных

На рис. 9.15 показано, как спроектировать оболочки данных, чтобы сделать API расширяемым.

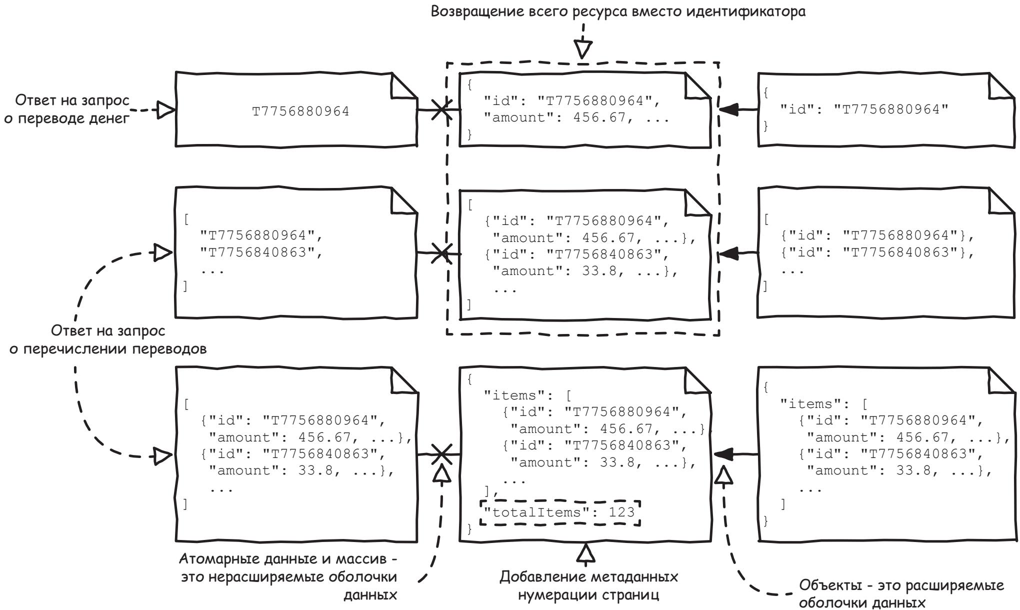



**Рис. 9.15. Выбор расширяемых оболочек данных**

Как вы думаете, что произойдет, если цель «Перевести деньги» просто вернет идентификатор денежного перевода, например "T775688964", в виде необработанной строки? Во-первых, потребители могут быть озадачены, потому что они получают ответ, заголовок Content-type которого имеет MIME-тип – `text/plain` вместо обычного `application/json` или `application/xml` используемого во всех других целях. Это неудобно, но они могут привыкнуть к этому… пока банковская бомпания не решит вернуть весь созданный ресурс, чтобы избежать множества последующих вызовов цели «Перечислить переводы». Вместо ответа `text/plain`, содержащего необработанную строку, теперь они получают ответ `application/json`, содержащий объект. Это критическое изменение. Если бы ответ был объектом, содержащим свойство `id` с самого начала, добавление других свойств перевода вообще не было бы проблемой. То же самое касается цели «Перечислить переводы», возвращающей список идентификаторов переводов в виде строк.

Если говорить о списках, возврат одного из них тоже не очень хорошая идея. Что произойдет, если нужно добавить метаданные для предоставления информации о нумерации страниц, например когда речь идет об общем количестве элементов? И снова критическое изменение. Чтобы избежать этого, нужно заключить список в свойство *items* внутри объекта.

Итак, как вы видите, все высокоуровневые данные (ресурсы в REST API) должны быть заключены в объект, чтобы обеспечить расширяемость и снизить риск критических изменений. А что насчет данных внутри этой оболочки? Как показано на рис. 9.16, вам следует остерегаться логических значений и предоставлять информативные данные.

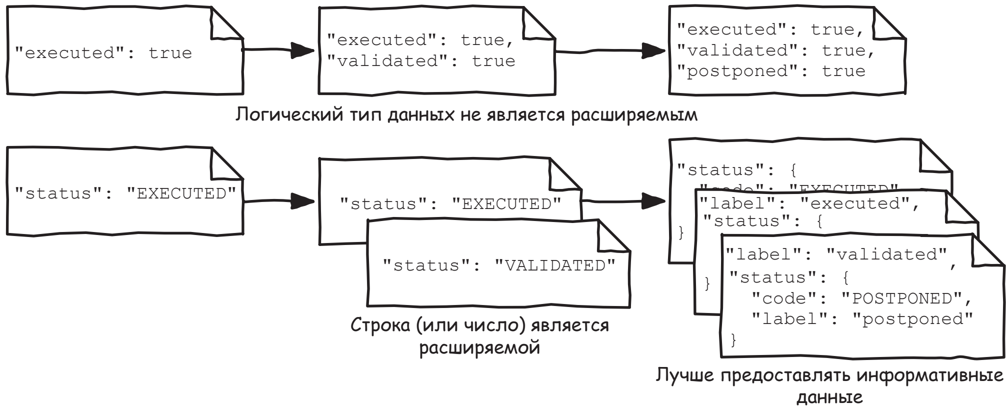



**Рис. 9.16. Разумный выбор типов и использование информативных данных**

Когда денежный перевод создается с использованием цели «Перевести деньги», он не выполняется незамедлительно. Чтобы предоставить информацию об этой транзакции, существует логическое свойство `executed`, которое имеет значение `true` при выполнении денежного перевода или `false` в противном случае. Что произойдет, если будет введено новое состояние?

Допустим, по какой-то причине некоторые денежные переводы нужно проверить, перед тем как они будут выполнены. Как справиться с этим? Можно добавить логическое свойство `validated`, чтобы обозначить это. Но что происходит, если вводится третье состояние `postponed`? Нужно ли добавлять еще одно логическое свойство? Добавление новых свойств в ответ не приводит к критическим изменениям, но потребители не будут знать об этих новых состояниях, если не обновят свой код.

Чтобы не добавлять несколько логических свойств состояния, вместо этого можно добавить одно свойство `status`. Это позволяет нам добавлять новые статусы по мере необходимости без добавления новых свойств. Данный статус может быть числом или строкой. Обратите внимание, что логическое значение менее расширяемо, чем число, которое менее расширяемо, чем строка. Но, как вы уже видели в разделе 9.1, добавление значений в перечисление может вызвать критические изменения. Если сделать состояние информативным объектом с кодом и легко интерпретируемой меткой, это может снизить риск.

Поэтому тщательно выбирайте типы своих свойств, чтобы обеспечить расширяемость, и всегда думайте о предоставлении информативных данных, чтобы снизить риск критических изменений. Все это работает только в том случае, если одного свойства достаточно для замены нескольких. На рис. 9.17 показано, что делать, когда это не так.

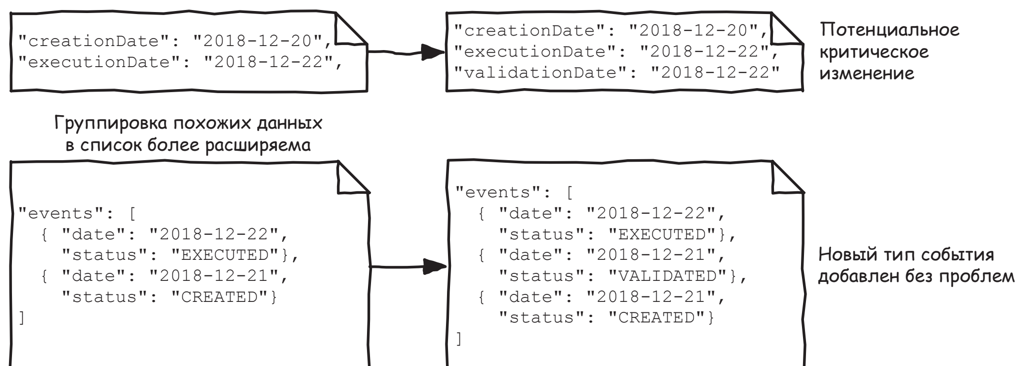



**Рис. 9.17.** Группировка похожих данных в списке

Денежный перевод имеет свойства `creationDate` и `executeDate`, соответствующие датам, когда он был создан и затем выполнен. Чтобы предоставить информацию о новом состоянии проверки, которое вы только что видели, можно добавить свойство `validationDate`. Но тогда возникнут те же проблемы, которые мы только что видели. Эти разные даты можно заменить свойством `events`, каждый элемент которого состоит из даты и состояния (например, `EXECUTED` – это дата выполнения). Добавить новые даты в список довольно просто; и, конечно же, свойство `status` может быть предоставлено в информативном формате.

Если свойства похожи, всегда смотрите, можно ли поместить их в список, возможно, используя информативные данные, что облегчит добавление элементов и уменьшит риск критических изменений. Говоря об информативных форматах, которые уменьшают риск внесения критических изменений: на рис. 9.18 показано, как можно использовать стандарты для проектирования расширяемых API.

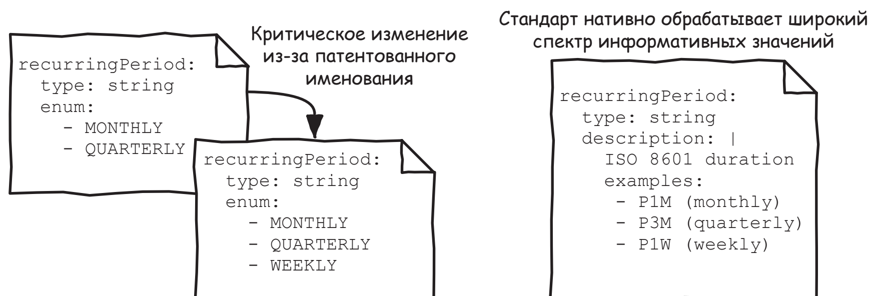



**Рис. 9.18.** Использование стандартов и более широкого диапазона информативных значений

Для описания периода регулярных денежных переводов можно использовать патентованное именование. Это могут быть уже готовые к использованию значения, такие как `MONTHLY` или `QUARTERLY`, но добавление нового значения, такого как `WEEKLY`, неизбежно приведет к критическим изменениям. Это также довольно жесткий подход. Что делать, если клиент хочет инициировать денежный перевод каждые 10 дней?

Нужно добавить новое значение. Использование формата продолжительности ISO 8601 может решить проблему. Он может описывать любую продолжительность, используя простой формат; например, `P1M` соответствует `MONTHLY`, а `P10D` – 10 дням.

Аналогичным образом вы уже видели преимущества использования кодов валют ISO 4217 для облегчения не только понимания, но и расширяемости. Если банковскому API нужно управлять новыми валютами, потребители смогут легко понять их, потому что они понимают стандарт ISO 4217. Таким образом, использование стандартов и более широкого диапазона информативных значений вместо законченного списка облегчает расширяемость и снижает риск ввода критических изменений.

### 9.3.2 Проектирование расширяемых взаимодействий

Закон Постеля гласит:

> *«Будьте консервативны в том, что вы делаете, и будьте либеральными в том, что вы принимаете от других».*[1]
>
> *Закон Постеля (Принцип надежности)*

Применительно к проектированию API принцип надежности можно понимать как «будьте последовательны в том, что вы возвращаете, и старайтесь избегать ошибок». Вы видели, как быть последовательным и обеспечивать расширяемость данных, возвращаемых API, в разделе 9.3.1, поэтому давайте сосредоточимся на ошибках.

Что касается данных об ошибках и последовательности в том, что мы возвращаем, можно применить то, что узнали, чтобы быть как можно более универсальными. В разделе 5.2.3 вы видели, что для обеспечения информативных ответных сообщений мы можем указывать тип ошибки, как показано в приведенном ниже листинге.

**Листинг 9.2. Информативное сообщение об ошибке**

```
{
  "errors": [
    { "source": "amount",
      "type": "MISSING_MANDATORY_ATTRIBUTE",
      "message": "Missing mandatory amount" }
  ]
}
```

Поскольку этот тип является универсальным, мы можем повторно использовать его для другого свойства, которое становится обязательным. Если бы это был тип `MISSING_AMOUNT`, мы не смогли бы использовать его повторно, и вместо этого были бы вынуждены ввести новый тип ошибки, который потребители не смогли бы интерпретировать без обновления своего кода. В целом, чем более универсальны значения `type`, тем более обширны сообщения об ошибках.

[1] Часто он звучит так: «Будьте консервативны в том, что вы отправляете, и будьте либеральными в том, что вы принимаете»

Что касается попыток избежать ошибок, что произойдет, если потребитель предоставит неизвестный параметр `test = 2` при запросе перечислить переводы? API может проявить строгость и вернуть сообщение об ошибке: «Извините, мы не понимаем параметр test». Предоставление информативного сообщения об ошибках удобно для потребителя, но API также может просто не учитывать этот неизвестный параметр, обработать запрос и вернуть результат. Это по-прежнему удобный для потребителя вариант, но он также и расширяемый. Если бы этот параметр `test` действительно существовал в предыдущей версии, необновленные потребители все равно могли бы отправить его, и их бы беспокоило, что их запросы теперь вызывают непредвиденную ошибку. Обратите внимание, что это работает, только если игнорирование параметра `test` не оказывает отрицательного побочного эффекта на стороне потребителя.

Давайте рассмотрим другие типы ошибок. Что должен делать банковский API, если потребитель отправляет параметр `pageSize=150` с запросом перечислить переводы, а максимальный размер страницы равен 100? По тем же причинам API должен возвращать не ошибку, а страницу, состоящую из 100 элементов. Затем, если в один прекрасный день (возможно, по соображениям производительности) максимальный размер уменьшится до 50, никого из потребителей это не побеспокоит. Метаданные нумерации страниц должны предоставлять всю необходимую информацию, чтобы потребители могли беспрепятственно использовать измененную цель; но, если это необходимо, предупреждающие метаданные можно добавить вместе с ответом, используя тот же формат, что и ошибки, для обозначения изменений, внесенных в запросы.

А что должен делать банковский API, если потребитель отправляет сумму в 15 000 долл., что превышает максимальную сумму перевода в 10 000 долл. (независимо от баланса исходного счета и привилегий владельца)? Должны ли мы инициировать денежный перевод в размере 10 000 долл. вместо 15 000? Очевидно, нет! Будучи проектировщиками API (и разработчиками), мы должны попытаться избегать ответных сообщений об ошибках, но не делать этого любой ценой.

Как видите, это дело реализации. Но, будучи проектировщиком API, вы должны будете определить политику в отношении ошибок и неизвестных или недопустимых параметров (параметров запроса, заголовков или свойств в теле). Вы не станете принимать во внимание проблему и использовать значение по умолчанию, чтобы уменьшить риск критических изменений? Или вы проявите строгость и будете возвращать ошибки, чтобы обеспечить дополнительную безопасность и способствовать точности со стороны потребителя (если их работа будет нарушена, они обновятся)? Ваш подход будет зависеть от контекста API и контекста каждой цели.

### 9.3.3 Проектирование расширяемых потоков

То, как вы проектируете каждую цель в потоке и сами потоки, будет влиять на расширяемость вашего API. Банковский API изначально был соз-

дан для мобильного приложения банковской компании. С помощью этого приложения конечные пользователи, осуществляющие денежные переводы, должны выбрать исходный счет, затем счет назначения, а затем указать сумму.

С точки зрения API это означает использование цели «Перечислить источники» для перечисления возможных источников перевода денег. После этого цель «Перечислить получателей» можно использовать для получения счетов и зарегистрированных бенефициаров, которые можно использовать в качестве получателя для перевода денег с использованием идентификатора определенного исходного счета. Наконец, цель «Перевести деньги» можно использовать для осуществления перевода с обозначенной суммой и идентификаторами исходного счета и счета назначения.

Теперь предположим, что некоторые сотрудники банковской компании решили создать инструмент для денежных переводов для бэк-офиса. Они были очень счастливы, когда обнаружили, что API для денежных переводов уже существует. В их реализации у них уже были номера исходного и конечного счетов, поэтому они просто вызывали цель «Перевести деньги», используя эти значения. К сожалению, все их вызовы завершались ошибкой «Неизвестный исходный счет».

Проведя расследование, они поняли, что цель «Перевести деньги» ожидала идентификаторы исходного счета и счета назначения, которые не были обычными номерами счетов. Они должны были вызывать цель «Перечислить исходные счета», чтобы найти идентификатор, соответствующий номеру их исходного счета, а затем следовать тому же потоку, что и в мобильном приложении. Какая жалость. Если бы поток не был так сфокусирован на сценарии использования мобильного приложения и если бы различные цели, вовлеченные в поток денежных переводов, использовали обычные номера счетов, все было бы намного проще.

Как видите, расширяемость в проектировании заключается не только в том, чтобы гарантировать, что изменения могут быть выполнены с низким риском ввода критических изменений. Расширяемость также означает, что API можно использовать в самых разных случаях, а не только в тех, для которых он был изначально создан.

Всегда старайтесь выходить за рамки конкретного варианта использования, с которым вы работаете, и следить за тем, чтобы проектируемые вами потоки, в особенности потоки пользовательского интерфейса, не коррелировали с конкретным процессом. Кроме того, постарайтесь проектировать каждый шаг так, чтобы его можно было использовать автономно. Выбор широко используемых входных и выходных данных, особенно идентификаторов, помогает добиться этого.

### 9.3.4 Проектирование расширяемых API

И последнее, но не менее важное: как обеспечить расширяемость на уровне API? Что произойдет, если банковский API будет расти, чтобы обеспечить несколько десятков целей, охватывающих различные темы, такие как информация о счете, подписка на банковские услуги, денеж-

ные переводы и управление личными финансами среди прочего? Очевидно, что произойдут критические изменения, даже если будут применены все принципы проектирования, которые мы видели до сих пор. Почему? Просто потому, что он большой!

Чем больше становится API, тем больше количество изменений и, следовательно, выше риск критических изменений. Это довольно просто понять, и решение очевидно: вместо того чтобы создавать большие API, нужно создавать API поменьше.

Однако не всегда так просто определить соответствующие группы целей, которые можно легко сочетать. Это не относится к проектированию API; это общая проблема, встречающаяся при проектировании программного обеспечения. Надеюсь, что, если вы помните раздел 7.2.3, у вас уже есть основы, которые могут помочь вам организовать цели и разбить API на более мелкие части. Вам также придется проанализировать каждую цель, которую вы добавляете в существующий API, чтобы оценить, должна ли добавляемая вами цель стать частью другого API, использующего те же принципы.

## *Резюме*

- Каждая модификация API должна быть тщательно спроектирована, чтобы избежать критических изменений, которые могут вызвать проблемы не только на стороне потребителя, но и на стороне поставщика.
- Проектировщикам API, возможно, придется смириться с предыдущими неудачными вариантами, чтобы избежать внесения благожелательных, но критических изменений.
- В зависимости от контекста критические изменения могут быть допустимы (например, закрытые API с потребителями под контролем организации).
- Управление версиями API – это вопрос проектирования + реализации + управление продуктом.
- Проектирование API с учетом расширяемости облегчает разработку модификаций, снижает риск критических изменений и способствует тому, что API можно будет использовать повторно.

# 10 Проектирование эффективного API для сети

**В этой главе мы рассмотрим:**

- проблемы передачи данных по сети с использованием веб-API;
- использование сжатия, кеширования и условных запросов;
- оптимизацию дизайна API, чтобы совершать меньше вызовов и обмениваться меньшим количеством данных.

Пока что мы занимались проектированием API, которые обеспечивают удобное, безопасное и расширяемое представление целей, имеющих смысл для потребителей и скрывающих внутренние проблемы. Но на самом деле мы научились проектировать идеальные лабораторные API, игнорируя большую часть контекста, в котором они используются, в особенности сетевого контекста.

Эффективность передачи данных по сети – важная тема, о которой должен знать любой проектировщик API. Эффективность передачи данных важна в нашей повседневной жизни. Когда вы общаетесь с кем-то непосредственно или с помощью мгновенных сообщений или по электронной почте, иногда вам нужна вся предыстория, а иногда вы хотите, чтобы человек, с которым вы разговариваете, сразу же понял, о чем идет речь, и рассказал только то, что вам нужно знать. Если вы получили всю информацию, но вам нужна была только определенная ее часть, вам придется тратить время, чтобы выслушать все это, или прочитать, что-

бы получить то, что вам нужно. Это может быть неприятно и даже может иметь серьезные последствия, например вы можете упустить какую-то возможность.

Выбор неправильного способа передачи информации может иметь негативные последствия в нашей повседневной жизни. То же самое относится и к API, которые обеспечивают неэффективную передачу данных по сети, как показано на рис. 10.1.

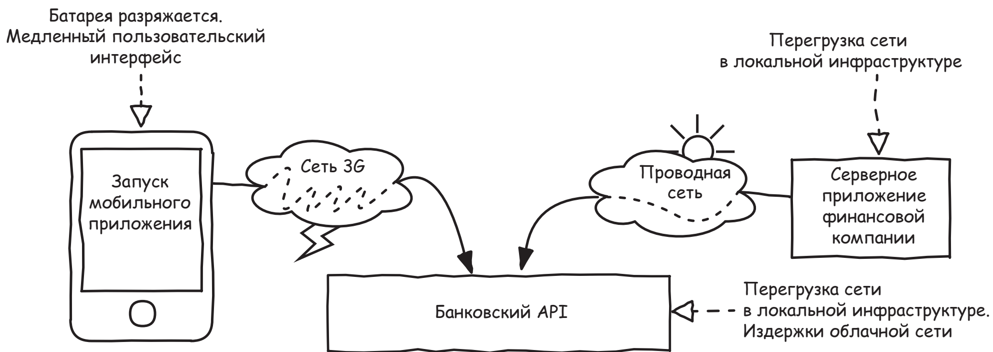



**Рис. 10.1.** Проблемы сети влияют на дизайн API

В мобильных телефонах неэффективные API-интерфейсы могут оказать существенное влияние на опыт разработчиков, затрудняя или даже делая невозможным их использование без замедления работы пользовательского интерфейса и разряда батареи устройства, что также отрицательно влияет на опыт пользователя. Даже при наличии современных устройств и сетей это менее важно, чем было когда-то для потребителей серверов. Такие неэффективные API могут оказать значительное влияние на использование пропускной способности сети. Это может быть проблемой для локальных инфраструктур с ограниченной пропускной способностью и также может влиять на поставщиков. Поставщики могут столкнуться с перегрузкой сети на локальных инфраструктурах или чрезмерно большими счетами на оплату облачной инфраструктуры.

Эффективность передачи данных по сети может быть серьезной проблемой, и вы, как проектировщик API, вносите в нее свой вклад. Проектировщики API должны иметь базовые знания о проблемах передачи данных по сети. Это включает в себя понимание того, как избежать неполадок в работе сети или решить их, используя преимущества базового протокола API или создавая эффективные API-интерфейсы.

## 10.1 Обзор проблем передачи данных по сети

Эта книга посвящена, в частности, удаленным API и веб-API. В разделе 1.1 вы увидели, что такие API позволяют потребителям взаимодействовать с поставщиком по сети. Мы можем забыть об этом, но, если возможности сети будут расти, в некоторых контекстах эффективность передачи данных по сети по-прежнему может быть важным вопросом по обе стороны провода.

Как проектировщик API, вы должны знать о проблемах передачи данных по сети, поскольку они могут повлиять на ваши проекты. Чтобы исследовать эту тему, мы проанализируем с точки зрения сети, как приложение Awesome Banking, мобильное приложение, работающее на мобильном телефоне, подключенном к не очень хорошей сети 3G, использует слегка измененную версию банковского API.

### 10.1.1 Подготовка сцены

Давайте подготовим сцену, чтобы начать. На рис. 10.2 показаны цели, предоставляемые банковским API, а на рис. 10.3 и 10.4 изображено, как все три экрана приложения Awesome Banking App используют их.

Банковский API

GET `/owners` Перечислить владельцев

Перечисляет владельцев счетов, к которым у конечного пользователя есть доступ. Один из них соответствует пользователю; другие могут быть членами семьи или партнерами. Возвращает обобщенную информацию по каждому из них

GET `/owners/{ownerId}` Чтение владельца

Получает подробную информацию о владельце

GET `/owners/{ownerId}/accounts` Перечислить счета владельца

Перечисляет счета владельца, к которым у конечного пользователя есть доступ. Возвращает обобщенную информацию по каждому из них

GET `/accounts/{accountId}` Чтение счета

Получает подробную информацию о счете

GET `/accounts/{accountId}/transactions` Перечислить транзакции по счету

Перечисляет транзакции по счету за последние три месяца. Возвращает обобщенную информацию по каждой из них

GET `/transactions/{transactionId}` Чтение транзакции

Получает подробную информацию о транзакции

**Рис. 10.2.** Банковский API

Экран панели инструментов (рис. 10.3) показывает всех владельцев, к чьим счетам у пользователя есть доступ, и выделяет счет, соответствующий пользователю. Для каждого владельца показано его звание и имя, общее сальдо на их текущих и сберегательных счетах, а также сумма всех транзакций для обоих типов счетов за последние три месяца (для простоты предположим, что все транзакции – это вывод средств).

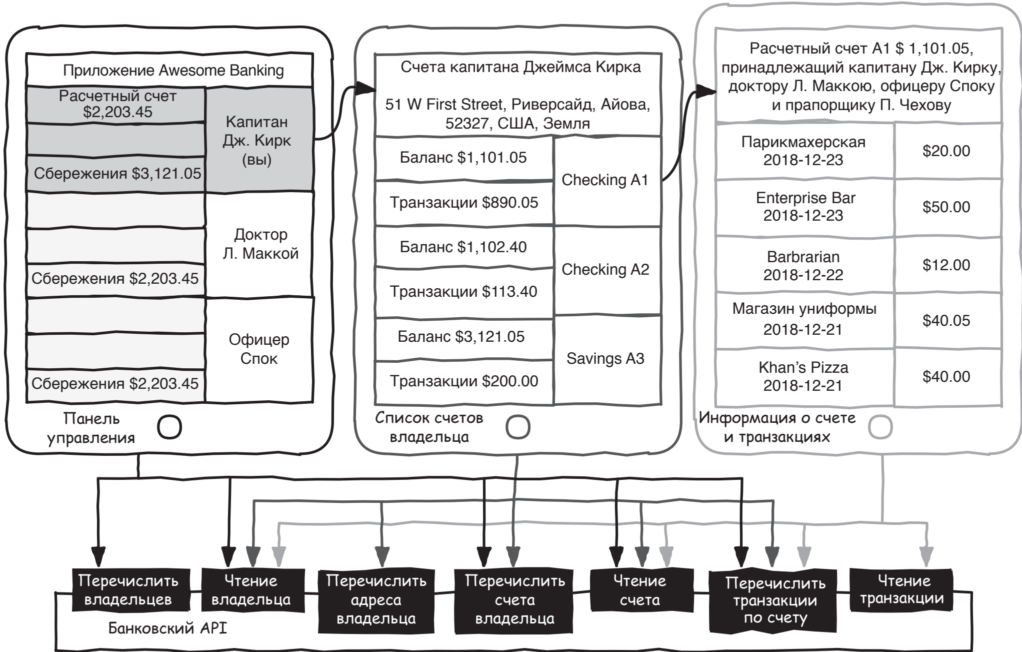



**Рис. 10.3.** Приложение Awesome Banking использует цели банковского API

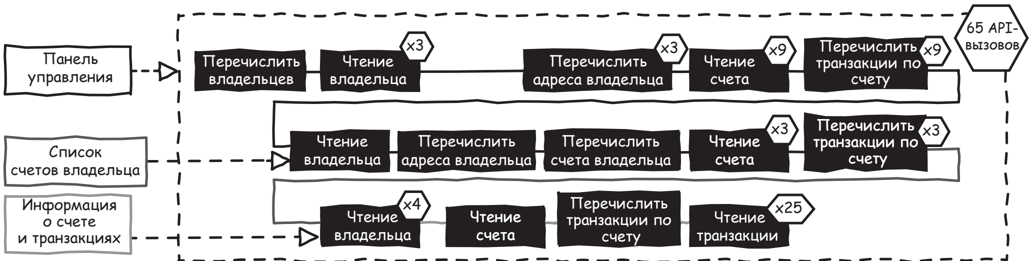



**Рис. 10.4.** Приложение Awesome Banking совершает слишком много вызовов

Для этого сначала перечисляются владельцы, чтобы получить их идентификаторы и имена, а также флаг пользователя, который указывает, какой владелец соответствует пользователю. Для каждого перечисленного владельца используется цель «Прочитать владельца», чтобы получить его звание. Затем используется цель «Прочитать счет», чтобы получить тип счета. И наконец, перечисляются транзакции для каждого счета, чтобы получить их суммы и сложить их все.

Когда пользователь нажимает на строку владельца, приложение переключается на экран списка счетов владельца. На этом экране отображаются звание, полное имя, домашний адрес и счета выбранного владельца. Для каждого счета показаны его тип (расчетный или сберегательный), идентификатор (например, A1,), баланс и сумма транзакции за последние три месяца.

Приложение начинается с использования цели «Прочитать владельца», чтобы получить имя и звание владельца счета. Затем перечисляются адреса для получения домашнего адреса владельца. После этого называются счета, чтобы получить идентификаторы и баланс каждого счета владельца, и используется цель «Прочитать счет» для получения типа счета. И наконец, перечисляются транзакции для каждого счета, чтобы получить суммы транзакций и сложить их.

Когда пользователь нажимает на строку счета, приложение переключается на экран подробной информации о счете и транзакциях. На этом экране показаны тип счета, идентификатор, баланс и владелец (владельцы). Все эти данные приложение получает, используя цели «Прочитать владельца», и «Прочитать счет». Затем перечисляются все последние транзакции ровно за последние три месяца с помощью цели «Перечислить транзакции», извлекая идентификатор транзакции и баланс по каждой из них. И наконец, только для последних 25 транзакций используется цель «Прочитать транзакции» для получения меток и дат. Приложение отображает пять самых последних транзакций (дополнительная информация о транзакциях будет получена, если пользователи прокрутят список).

Тут много API-вызовов – 65, если быть точным. Ваше сознание, вероятно, уже говорит вам, что здесь что-то не так. И в самом деле, пользователи приложения Awesome Banking жалуются на то, что оно слишком медленное, разряжает батарею и потребляет слишком много трафика. И на другой стороне не лучше! Банковская компания, предоставляющая этот API, весьма обеспокоена счетами от своего облачного провайдера.

### 10.1.2 Анализ проблем

Почему пользователи приложения Awesome Banking жалуются? Почему это приложение медленное и неэффективное? И почему банковская компания обеспокоена счетами от своего облачного провайдера? Все сводится к числу и частоте сетевых вызовов и объему обмена данными. В этом разделе мы проанализируем эти проблемы.

Пожалуйста, имейте в виду, что то, что вы видите здесь, не является точным отражением реальности; это упрощенное объяснение конкретного случая, в котором все возможные проблемы были грубо выделены, чтобы предоставить обзор проблем передачи данных по сети, которые проектировщики API должны учитывать при проектировании своего API. Поведение мобильного приложения и дизайн API на самом деле довольно идиотские, и, очевидно, никто никогда не станет создавать такое убожество – я надеюсь!

Давайте начнем с того, что разложим на части сетевой вызов API, осуществляемый по мобильной сети. На рис. 10.5 показано, что происходит, когда в приложении Awesome Banking перечисляются транзакции за последние три месяца.

Первый этап – подключение к серверу, на котором размещен банковский API. Здесь много различных действий, включающих радиоантен-

ну телефона, запуск низкоуровневой передачи данных по сети и шифрование.

**Рис. 10.5.** **Разложим на части API-вызов, сделанный по мобильной сети**

Предположим, что на это всегда уходит около 300 мс, но мы знаем, что в зависимости от типа мобильной сети, качества радиосигнала и того, что произошло до вызова, может потребоваться до нескольких секунд. Это время часто называют *задержкой*.

Как только соединение установлено, можно отправить API-запрос. Поскольку это простой запрос `GET /accounts/{accountId}/transactions`, его размер составляет всего около 100 байт, поэтому его загрузка или отправка на сервер API занимает гораздо меньше чем 1 мс. Когда сервер получает запрос, он должен обработать его, загрузить транзакции из базы данных и сгенерировать документ в формате JSON. Предположим, что на это уходит 20 мс, но фактическое время зависит от того, что запрашивается, и от возможностей сервера.

Наконец, четвертый этап состоит в загрузке ответа сервера. Предположим, что размер сгенерированного документа в формате JSON составляет около 32 Кб, поэтому его загрузка занимает 200 мс со скоростью 160 Кб/с (что рекламируется как 1,3 Мб/с). Как и задержка, *скорость загрузки* сильно зависит от типа сети и качества радиосигнала. Весь запрос занимает 520 мс. Немного долго, но, учитывая неблагоприятные условия сети, мы должны с этим смириться. Один этот запрос не так уж и плох, но давайте посмотрим, что произойдет, когда приложение Awesome Banking фактически использует API для заполнения экрана панели управления в простейшем случае: пользователь, соответствующий одному владельцу, у которого есть один счет.

На рис. 10.6 показано, какие API-вызовы выполняются и как это делается. Чтобы было проще, мы будем принимать как должное, что пропускная способность сети составляет 160 КБ/с, задержка всегда – 300 мс, отправка запроса занимает 0 мс, а серверу для обработки запроса всегда требуется 20 мс.

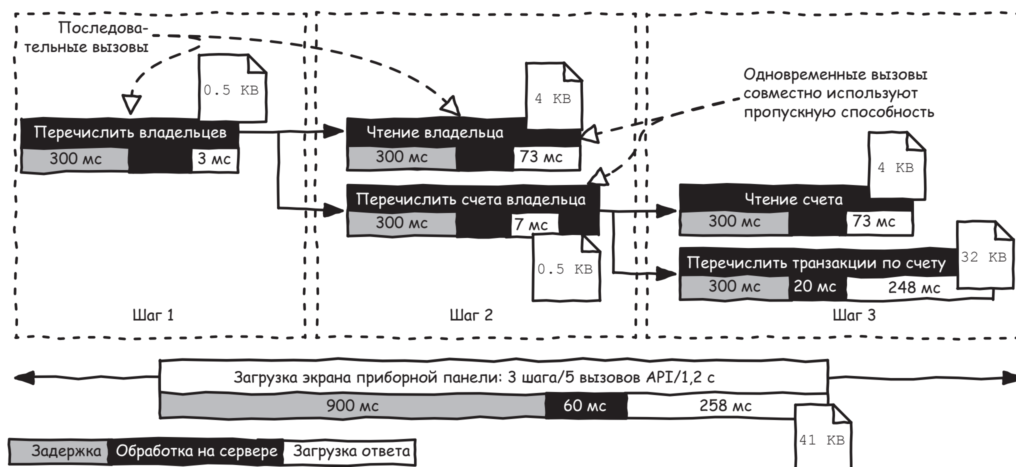



**Рис.10.6. Базовый вариант с пользователем, имеющим доступ к одному владельцу, которому принадлежит один счет**

Для загрузки данных, отображаемых на экране панели инструментов, в приложении Awesome Banking App перечисляются владельцы, чьи счета пользователь имеет право просматривать; здесь оно возвращает единственного владельца, соответствующего пользователю. Затем оно читает подробную информацию этого владельца и параллельно перечисляет счета владельца. После того как приложение получило список, который в данном случае состоит из одного счета, оно считывает подробную информацию об этом счете и параллельно перечисляет транзакции.

Вся цепочка вызовов API состоит из пяти вызовов, распределенных в три этапа благодаря параллельным вызовам. К сожалению, на это уходит 1,2 с, что превышает 500 мс, верхний предел допустимого *диапазона задержки* (время, выше которого человеческий мозг допускает задержку). И это в простом варианте использования! Теперь давайте посмотрим, как это происходит в более сложном случае. На рис. 10.7 показано, что происходит с другим пользователем, у которого есть доступ к четырем владельцам и их восьми счетам.

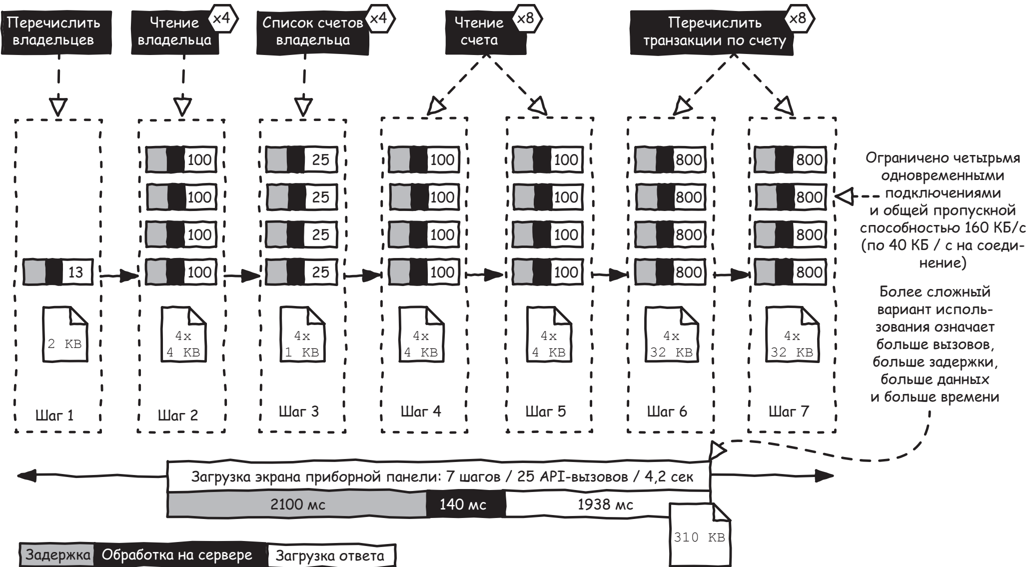



**Рис. 10.7. Сложный вариант с пользователем, у которого имеется доступ к четырем владельцам и их восьми счетам**

Приложение запускается снова, перечисляя владельцев, но на этот раз их четыре. Поскольку мобильное приложение ограничено операционной системой четырьмя одновременными HTTP-соединениями, оно считывает подробную информацию для каждого из владельцев счета, используя четыре параллельных запроса; после этого перечисляются их счета, также параллельно. Обратите внимание, что, если вся пропускная способность сети составляет 160 Кб/с, каждый параллельный запрос выполняется на четверть от этого показателя, т. е. со скоростью 40 Кб/с.

Затем приложение считывает подробную информацию для восьми счетов владельцев и их транзакций в четыре этапа из четырех параллельных запросов. Таким образом, в этом случае 25 API-вызовов выполняется в семь последовательных шагов, на что уходит 4,2 с, и мы получаем 310 Кб данных. Это на 20 вызовов больше и примерно в 7,5 раз больше данных, чем в предыдущем случае, и вся цепочка занимает в 3,5 раза больше времени. Это очень большая задержка, и, даже если приложение показывает счетчик, пользователям определенно будет скучно, прежде чем информация загрузится на экране.

У каждого экрана, отображаемого приложением, могут быть проблемы одного и того же типа. Представьте себе, что происходит на экране счета, когда приложение должно совершать отдельные вызовы, чтобы получить подробную информацию о каждой транзакции. Дальше – хуже. Пользователи могут перемещаться по приложению, чтобы увидеть все свои данные, и это может привести к другим проблемам, как показано на рис. 10.8.

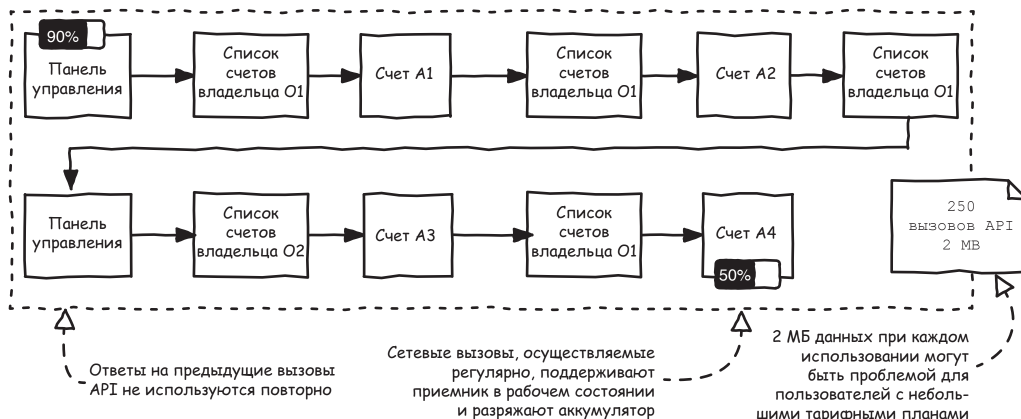



**Рис. 10.8.** Как примерно выглядит навигация по приложению Awesome Banking

Команда приложения Awesome Banking изучила его аналитические данные, и, кажется, у обычного пользователя имеется доступ к двум владельцам, а у каждого владельца в среднем по два счета. Команда также определила, что эти средние пользователи обычно перемещаются по всем своим данным, когда пользуются приложением. К сожалению, приложение не использует повторно ответы на предыдущие вызовы API, поскольку не знает, можно ли повторно использовать данные. Поэтому новые вызовы API выполняются на каждом экране, даже если данные уже получены. Это приводит к тому, что выполняется много API-вызовов (в среднем около 250 на сеанс), и объем данных, которыми обмениваются, значителен (около 2 Мб). В зависимости от того, как часто пользователи проверяют свои счета, приложение Awesome Banking может использовать от 60 до 90 Мб данных в месяц.

В настоящее время использование данных почти не является проблемой для пользователей в большинстве стран, но это не относится ко всем пользователям в мире. А для пользователей, путешествующих за границу, каждый килобайт может иметь значение, потому что в этом случае данные стоят очень дорого. Но, помимо тарифного плана, больше всего раздражает то, что все вызовы API, сделанные на каждом экране, разряжают батарею устройства, потому что радио работает в течение более длительного периода времени. Очевидно, что количество вызовов, их длительность и объем данными, которыми обмениваются, могут вызывать беспокойство со стороны потребителя, но то же самое можно сказать и о поставщике.

Допустим, банковский API размещен на облачном сервисе Barbrarian (который изначально продавал книги онлайн, но это уже другая история). API реализован с использованием совершенно нового внесерверного сервиса под названием «Функции». Каждая цель кодируется отдельно как функция, и вам больше не нужно беспокоиться о серверах, приложениях и масштабировании. Всем занимаются системы: когда поступают вызовы, запускается соответствующая функция. Стоимость услуги основана на количестве принятых вызовов, времени обработки

и объема исходящих данных. Это означает, что чем больше вызовов API, и чем они больше, тем больше банковской компании придется платить. Таким образом, оптимизируя все, что может быть важным и даже жизненно важным.

Это только два примера. API могут иметь контексты, отличные от тех, которые мы только что видели, но показатели эффективности сети обычно будут зависеть от следующих факторов: скорость, объем данных и количество вызовов. Как проектировщики API, мы должны знать об этом и пытаться найти баланс между потребностью в эффективности и идеальным дизайном. Далее мы рассмотрим, как можно найти этот баланс и оптимизировать передачу данных по сети на уровне протокола.

## 10.2 Обеспечение эффективности передачи данных по сети на уровне протокола

Оптимизация передачи данных по сети начинается на уровне протокола. Используя преимущества базового протокола, можно создать эффективный API без необходимости слишком много работать с идеальным дизайном. Для API на базе протокола HTTP активация сжатия и постоянных соединений может уменьшить объем данных и задержку.

Активация *кеширования* (информирования потребителей о том, могут ли они сохранить ответ для повторного его использования и насколько) и *условных запросов* (позволяющим потребителям проверить, достаточно ли свежи данные, которые у них имеются, чтобы избежать их повторного получения), может уменьшить не только объем данных, но и количество запросов.

### 10.2.1 Активация сжатия и постоянных соединений

Для API на базе протокола HTTP есть две довольно распространенные оптимизации, которые могут быть выполнены без влияния на дизайн: активация сжатия и постоянных соединений. После активации сжатия на сервере банковского API 310 Кб данных, полученных для четырех владельцев и их восьми счетов во втором нашем варианте, можно уменьшить до размера менее 2 Кб. Для приложения Awesome Banking, работающего на мобильном телефоне, подключенном к не очень хорошей сети 3G, это означает меньшее количество данных и более короткие вызовы. Меньшие ответы будут загружаться быстрее, и приложение не будет использовать трафик конечного пользователя. Большинство – если не все потребители, использующие стандартную библиотеку HTTP – может воспользоваться этой функцией без необходимости менять что-либо в своем коде.

Эта модификация сервера также приносит пользу и потребителю. Если банковский API размещен в локальной инфраструктуре, меньшее количество данных означает меньший риск перегрузки сети, поскольку общая используемая пропускная способность ниже и сетевые подключения будут открыты в течение меньшего времени. Для облачных инфраструктур это просто означает не такие большие счета.

После того как на сервере банковского API будут активированы постоянные соединения, сценарий использования, где требовалось 25 вызовов за семь шагов с общим временем в 4,2 с, можно уменьшить до 2,4 с, удалив задержку 6×300 мс. Когда постоянные HTTP-соединения активированы, только первый API-вызов будет испытывать задержку соединения; последующие вызовы осуществляются с использованием одного и того же соединения. Соединение остается открытым в течение определенного количества вызовов или времени, определяемого конфигурацией сервера.

Переход на HTTP/2 также может быть вариантом; эта версия протокола HTTP предлагает эффективные постоянные соединения, параллельные запросы и двоичный транспорт, а самое главное – она обратно совместима с HTTP/1.1! С точки зрения проектирования API использование HTTP/2 имеет примерно тот же эффект, что и активация сжатия и постоянных соединений. Эта версия прозрачна и не требует изменений. В обоих случаях идея состоит в том, чтобы уменьшить задержку и объем данных, которыми обмениваются, а следовательно, и длину вызовов API.

**ПРИМЕЧАНИЕ.** Если проектировщики API не являются техническими руководителями проекта, архитекторами или разработчиками, которые, как правило, обрабатывают такие виды оптимизации, они должны, по крайней мере, знать о возможных оптимизациях протоколов (например, таких как постоянное соединение или сжатие), чтобы предлагать решения возможных проблем производительности сетевого взаимодействия.

Для разработчиков API важно уточнить у команды специалистов, занимающихся работой над API, рассматривались ли такие оптимизации до изменения существующего проекта из-за проблем, связанных с производительностью передачи данных по сети. В идеале такие оптимизации должны были проводиться с самого начала, и оптимизация обмена данными для эффективности может иметь важные последствия для дизайна. В зависимости от контекста было бы разумно использовать разные типы API и даже разные способы передачи данных.

### 10.2.2 Активация кеширования и условных запросов

Второй способ оптимизировать передачу данных – вообще не обмениваться данными или, по крайней мере, делать это реже. В разделе 10.1.1 вы видели, что приложение Awesome Banking делало много ненужных вызовов, потому что оно не использовало повторно ответы на предыдущие вызовы. В частности, для создания экрана панели мониторинга приложение Awesome Banking загрузило большое количество данных, которые можно было использовать повторно при отображении списка счетов владельца. На рис. 10.9 показаны API-вызовы, выполненные без кеширования ответов и с ним.

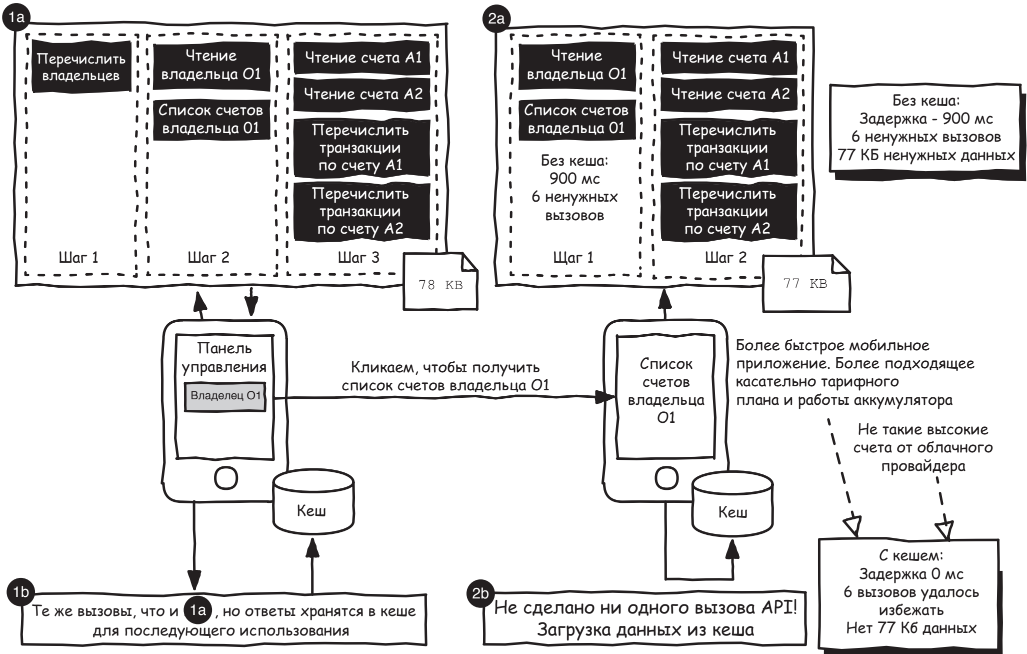



**Рис. 10.9.** Сокращение количества вызовов с помощью кеширования

Сценарий в верхней части рисунка показывает абсолютно неоптимизированное приложение Awesome Banking, которое повторно не использует ответы на вызовы API. В сценарии внизу рисунка приложение кеширует ответы на все запросы, сделанные на экране панели мониторинга. Это позволяет избежать шести вызовов при отображении экрана списка счетов владельца, поскольку оно может получать необходимые данные из кеша, что хорошо для приложения и его пользователей: приложение быстрее реагирует при переходе от информационной панели к списку счетов и использует меньше ресурсов сети и аккумулятора. Это также хорошо для банковской компании, которая предоставляет API; избегая шести ненужных вызовов API в этом случае, трафик к банковскому API сокращается на 46 % по вызовам и на 50 % по объему данных.

Как видите, кеширование может очень помочь, если нужно сделать передачу данных по сети более эффективной. Это то, что сделал бы любой разработчик мобильных приложений, даже не задумываясь об этом. Потребители могут кешировать любые ответы API, но API может помочь им делать это правильно и эффективно. Проектировщик API отвечает за определение того, какие данные должны кешироваться и как долго. На рис. 10.10 показано, как это можно сделать с помощью протокола HTTP.

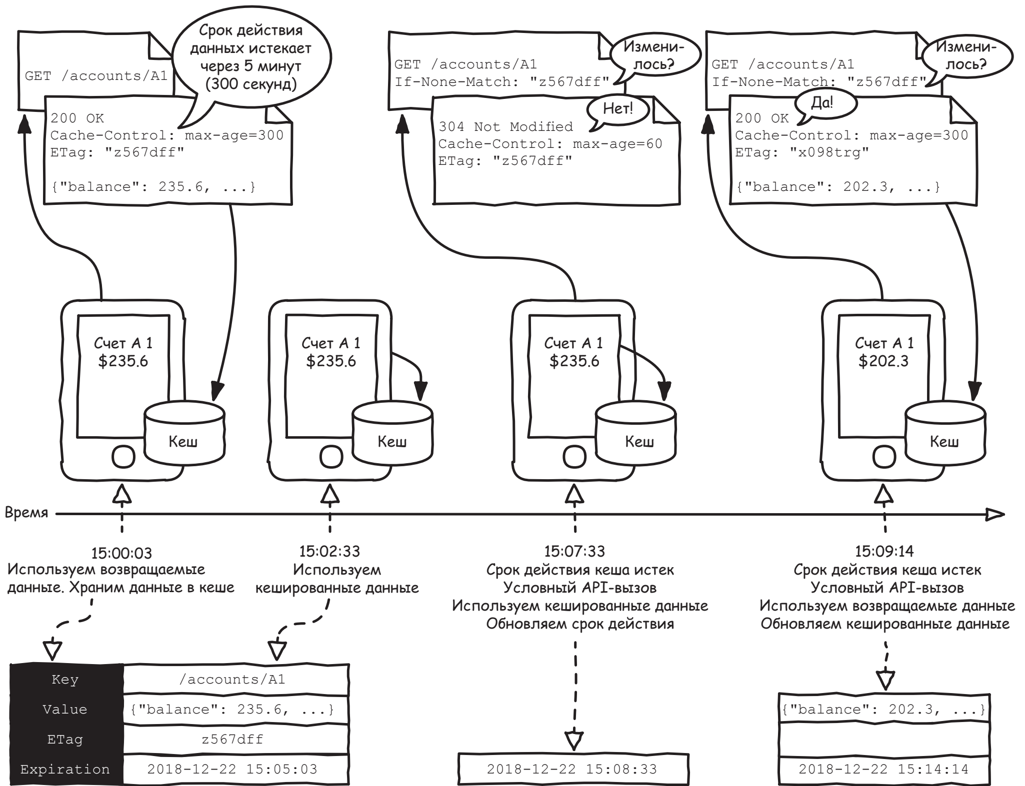



**Рис.10.10. Использование функций кеширования протокола HTTP**

Когда приложение Awesome Banking получает информацию о первом счете с помощью HTTP-запроса `GET /accounts/A1`, API возвращает ответ `200 OK` вместе с данными и HTTP-заголовками `Cache-Control` и `ETag`. Значение заголовка `Cache-Control` – `max-age=300`, а это означает, что этот ответ можно кешировать на 300 с (5 мин). Поэтому, если приложение должно снова показать данные этого счета в течение следующих пяти минут, оно будет использовать кешированный ответ вместо вызова API.

По истечении пяти минут срок действия кеша истекает; следовательно, приложение должно будет сделать еще один HTTP-запрос `GET /accounts/A1`, если ему снова понадобится информация о первом счете. Однако, вместо того чтобы отправлять точно такой же запрос, как в первый раз, он может отправить условный запрос, который в основном гласит: «Дайте мне данные по счету A1, только если они были изменены». Это делается с помощью значения `z567dff` `ETag`, возвращаемого первым вызовом. Заголовок `If-None-Match: "z567dff"` отправляется вместе со вторым запросом.

Когда сервер банковского API получает запрос, он использует значение `z567dff`, чтобы проверить, были ли изменены данные счета. Это значение описывает состояние ресурса, который был отправлен с первым запросом. Это может быть хеш данных, число или номер версии или

любое другое значение, которое позволяет серверу узнать, какая версия ресурса была предоставлена ранее. Потребителям не нужно знать, каково это значение на самом деле.

Если данные не были изменены, сервер возвращает ответ `304 Not Modified` без каких-либо данных и таким образом избегает ненужной загрузки данных. Приложение обновляет дату истечения срока действия кеша, которая может отличаться от первого вызова, благодаря заголовку `Cache-Control`, возвращаемому с ответом с кодом `304`. Заголовок `ETag` не изменяется, поскольку данные не изменились.

Тот же запрос, сделанный позже, может получить ответ `200 OK`, который говорит: «Да, данные были обновлены, вот они». Этот ответ содержит обновленные данные вместе с заголовками `Cache-Control` и `ETag`. В нашем примере значение `Cache-Control` равно `max-age=300`, как и раньше, а значение `ETag` теперь равно `x098trg`. Его значение изменилось, потому что изменились данные ресурса.

Если приложение только хочет узнать, изменились ли данные первого счета, но на самом деле не хочет загружать новые данные, оно может отправить запрос `HEAD /accounts/A1` вместо `GET /accounts/A1`. HTTP-метод `HEAD` идентичен `GET`, за исключением того факта, что сервер возвращает не содержимое ресурса, а только заголовки.

### Ограничения REST: кеширование

Архитектурный стиль REST гласит, что ответ на запрос должен указывать, можно ли его сохранить (чтобы клиент мог повторно использовать его вместо повторного вызова) и насколько. Это определенно необходимо для любого API, чтобы гарантировать, что потребители не будут делать ненужные вызовы, а также чтобы поставщики не отправляли слишком много ресурсов просто так, а возвращали данные, которые уже есть у потребителей. Здесь мы видели только некоторые возможности кеширования HTTP; если вы хотите узнать больше, посетите страницу по адресу https://tools.ietf.org/html/rfc7234.

Возвращая и принимая некие метаданные, а также предоставляя способы получения только этих метаданных без данных, API может активировать кеширование своих ответов и предлагать условные запросы, которые значительно оптимизируют передачу данных по сети за счет сокращения числа вызовов и объема возвращаемых данных. Кеширование также гарантирует определенную актуальность и точность данных.

Если выбранный вами тип протокола или API предоставляет эти функции изначально, не стесняйтесь их использовать. Однако имейте в виду: только потому, что ответ от API можно кешировать, это не означает, что потребители фактически делают это. Кроме того, функции кеширования не всегда могут быть доступны (например, на момент написания этой книги фреймворк gRPC не предоставляет таких возможностей). В таких случаях, если вы решите, что кеширование действительно важно, у вас есть два варианта. Первый – пересмотреть ваш выбор протокола и типа

API; проверьте, действительно ли было лучше использовать их в зависимости от контекста. Второй вариант заключается в воссоздании эквивалентных функций в вашем API, рискуя предоставить решение, которое является настолько нестандартным или необычным (или неэффективным), что потребители его вообще не используют.

При проектировании REST API на базе HTTP кеширование кажется относительно простым; нам нужно лишь добавить соответствующий HTTP-метод, код состояния и заголовки в описание API. Но это еще не все. Откуда берется срок действия кеша в 5 мин? Почему не 15 мин или 2 дня? Почему кеширование вообще разрешено? Хитрость кеширования заключается в том, что его возможности должны оцениваться для каждой цели, а говоря более конкретно – для каждого свойства, возвращаемого целью.

### 10.2.3 Выбор политики кеширования

Данные, возвращаемые, когда потребитель запрашивает информацию о счете, могут быть датой его создания, именем и балансом. Дата создания никогда не меняется. Имя может измениться, но это случается редко. И напротив, баланс обновляется всякий раз, когда происходит транзакция. Поскольку баланс – это свойство, которое будет меняться чаще других, его время жизни определяет продолжительность кеширования данных банковского счета, возвращаемых этой целью. Итак, как мы можем определить правильное значение продолжительности кеширования? По-разному.

Как долго данные могут быть кешированы, зависит от того, насколько часто они обновляются. Вначале банковская компания обновляла только список операций по счету и, следовательно, баланс счета несколько раз в день. Таким образом, длительное кеширование продолжительностью один час было практически осуществимо. Но теперь компания усовершенствовала свою систему: транзакции с другими банками по-прежнему обрабатываются партиями по несколько раз в день, но теперь все внутренние транзакции обрабатываются в режиме реального времени. Следовательно, чтобы предоставить точные данные, подходящая продолжительность кеширования должна определяться не только тем, как работает банковская система, но и тем, как люди на самом деле используют свои банковские счета.

Банковская компания определила, что статистически пятиминутное кеширование предлагает неплохой баланс между точностью и эффективностью при получении информации о счете.

В ближайшем будущем, когда все коммуникации между банками будут осуществляться в режиме реального времени и люди всегда будут получать информацию по своим счетам таким образом, кеширование данных может оказаться вообще невозможным. В этом случае значение заголовка `Cache-Control` будет равно `"0"`, но, по крайней мере, все еще будет возможно делать условные запросы, используя значение `ETag`, чтобы избежать загрузки неизменных данных.

То, как данные могут быть кешированы, также может зависеть и от других вещей. Например, представление неточного баланса, даже с помощью

стороннего приложения, может вызвать проблемы с юридической точки зрения или с точки зрения безопасности. Поэтому в документации банковского API может быть указано, что потребители должны использовать актуальные данные. Как и в случае информации о балансе в реальном времени, при таком сценарии API предоставит заголовок Cache-Control со значением «0», но использовать условные запросы по-прежнему можно.

Правовые соображения или соображения безопасности также могут помешать потребителям хранить данные, или же может существовать промежуточный уровень, где разрешено кеширование, но не хранение. В этом случае значение заголовка `Cache-Control` может быть равно `"60", no-store"`. Это означает, что данные могут кешироваться в энергозависимом ЗУ в течение 60 с, но не могут храниться в нем (`no-store`).

Говоря в двух словах, как вы уже видели в разделе 8.4.1, вам может понадобиться совет от специалистов по безопасности и юридического отдела при проектировании своего API. Таким образом, хотя активировать условные запросы довольно просто, для того чтобы быть эффективным и точным, требуется определенная работа, чтобы выяснить, возможно ли на самом деле кеширование, и если да, то какую продолжительность кеширования лучше всего выбрать.

Эти виды оптимизации могут быть встроены с самого начала, но их также можно реализовать, когда API уже используется, без особого влияния на контракт интерфейса, не прибавляя проектировщикам API лишней работы. Но на проектировщиках лежит гораздо большая ответственность, нежели простая проверка того, активированы ли сжатие и постоянные соединения и действительно ли потребители используют кеширование. Сама конструкция API может быть причиной неэффективного обмена данными по сети между потребителями и поставщиками.

## 10.3 Обеспечение эффективности передачи данных по сети на уровне дизайна

То, что обмен данными между потребителями и поставщиками можно оптимизировать на уровне протокола, не означает, что можно проявлять небрежность на уровне дизайна. По сути, дизайн API определяет количество вызовов, которые потребителям необходимо совершить для достижения своих целей, и объем данных, которыми обмениваются потребители и поставщики. Применяя то, что вы узнали в предыдущих главах, вы можете проектировать API, которые обеспечивают точные цели, оптимизированную детализацию и организацию данных, а также достаточную гибкость для обеспечения эффективности передачи данных. Напомню, что мы будем работать с банковским API, текущее состояние которого показано на рис. 10.11.

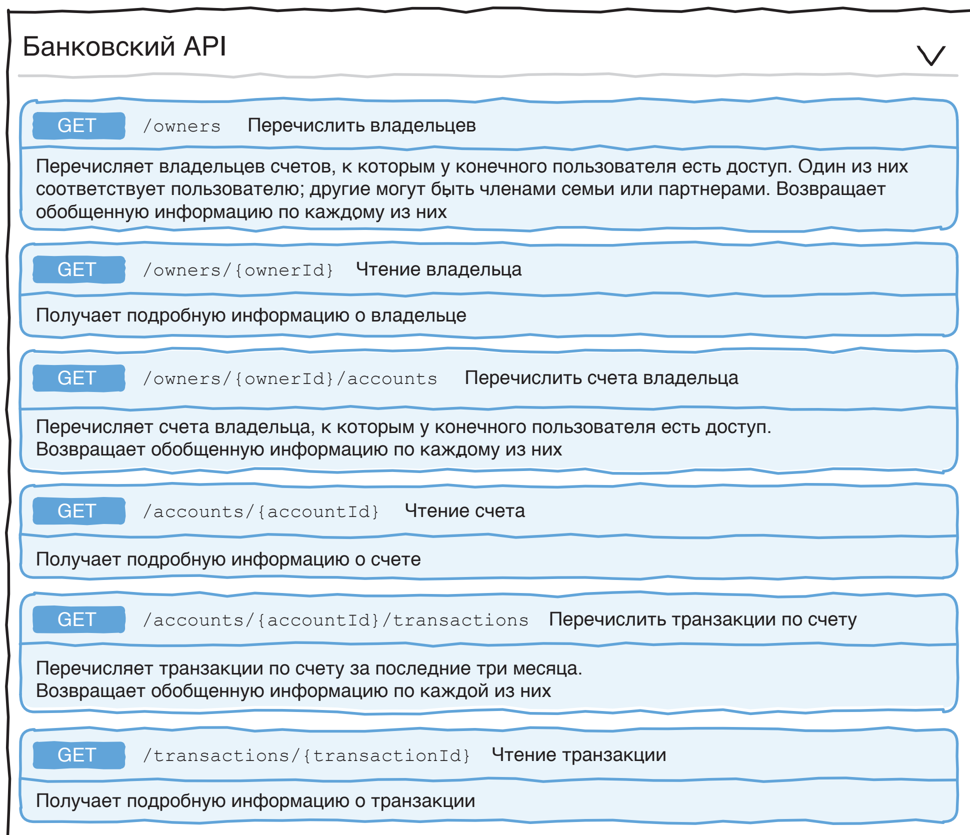



**Рис. 10.11.** Цели банковского API

Этот вариант довольно прост для понимания и, по крайней мере, на первый взгляд он кажется актуальным и хорошо организованным. Но с точки зрения эффективности передачи данных по сети он далеко не идеален. Давайте рассмотрим стратегии, которые можно использовать для оптимизации числа вызовов и объема данных, которыми обмениваются, когда банковский API используется приложением Awesome Banking или любым другим потребителем.

## 10.3.1 Активация фильтрации

Предоставление параметров фильтрации – хороший способ уменьшить объем обмена данными, поскольку это позволяет потребителям получать именно то, что им действительно нужно. Цель «Перечислить транзакции» всегда возвращает транзакции за последние три месяца, отсортированные от самой последней (наиболее поздней) до самой первой (наименее поздней). В контексте приложения Awesome Banking такая глубина данных необходима при отображении совокупной суммы сумм транзакций на экранах панели мониторинга и списка счетов, даже если пользователи, вероятно, так и не будут просматривать все эти транзак-

ции. Благодаря возможностям кеширования мы можем повторно использовать этот огромный список транзакций на экране счетов. Но, к сожалению, когда срок действия кеша истекает, приложение должно снова перезагрузить все транзакции за три месяца, даже если была добавлена только одна. Похоже, что эта цель была специально спроектирована для особых нужд экранов панели инструментов и списка счетов приложения Awesome Banking, но результат не очень эффективен.

В разделе 6.2.3 вы узнали, что всегда предоставлять все данные может быть не очень хорошей идеей, поскольку потребителям могут не понадобиться все данные во всех ситуациях. Использование фильтрации делает API более удобным и эффективным, позволяя потребителям запрашивать только те данные, которые им действительно нужны, – и это то, что нам здесь крайне необходимо. Каждый сохраненный байт повышает эффективность передачи данных в контексте небезопасной сети.

Исходя из того, что вы узнали, можете добавить параметры запроса `page` и `size`, чтобы обеспечить функции нумерации страниц на основе смещений, но учтите, что цель все равно может возвращать транзакции по всем трем месяцам без этих параметров, чтобы сохранить обратную совместимость, если API уже используется. Похоже, что экран счета может эффективно использовать эту функцию. При отправке запроса `GET /accounts/A1/transactions?page=1&size=25` возвращаются только последние 25 транзакций по счету A1. Если пользователи прокручивают страницу вниз, приложение может запросить следующую страницу с помощью запроса `GET / accounts/A1/transactions?page=2&size=25`. Но что произойдет, если между этими двумя запросами произойдут новые транзакции? Некоторые транзакции с первой страницы будут перемещены на вторую, поэтому второй запрос вернет уже полученные транзакции. Потребитель должен проверить, получил ли он уже транзакцию, и проигнорировать дубликаты.

Такое может происходить не так уж часто, но это может привести к предоставлению неточной информации, что недопустимо для банковской компании. Такой способ разбиения транзакций на страницы не подходит для этого случая и контекста, и это не решает проблему перезагрузки транзакций по всем трем месяцам на других экранах. Итак, какие же фильтры можно предоставить для решения этой проблемы?

Потребители, кем бы они ни были, в основном должны иметь возможность получать любые транзакции до или после выбранной транзакции, чтобы получить именно те данные, которые им нужны. На рис. 10.12 показано, как это можно сделать с помощью пагинации на основе курсора.

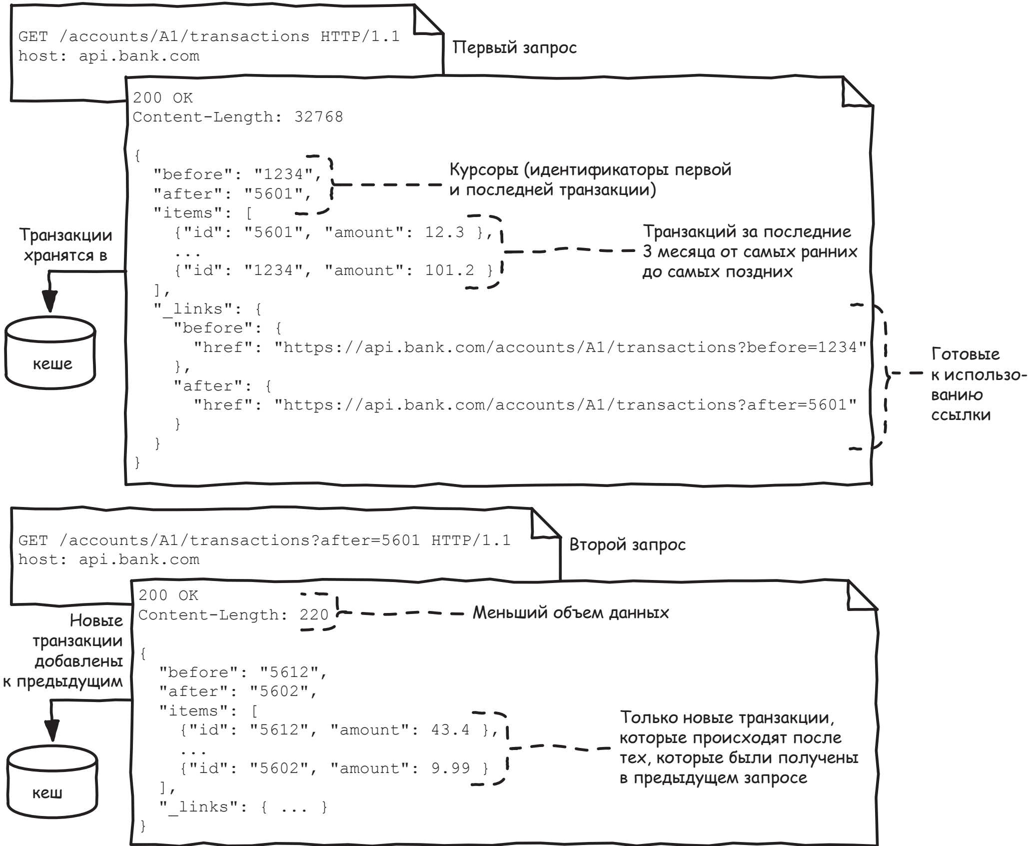



**Рис. 10.12.** Пагинация на основе курсора для получения транзакций

Приложение Awesome Banking по-прежнему отправляет первоначальный запрос на получение транзакций за последние три месяца с использованием запроса `GET /accounts/{accountsId}/transactions`, но теперь ответ содержит метаданные пагинации. Свойства `before` и `after` – это значения курсора, которые можно использовать для получения транзакций до или после извлеченного набора. Их значения – это идентификаторы первой (самой последней) и последней (самой ранней) транзакций из набора соответственно. Чтобы извлечь только транзакции, которые произошли, после того как был получен текущий набор транзакций, приложение должно отправить второй запрос:

```
GET /accounts/{accountId}/transactions?after=
➥ {latestTransactionId}
```

Значение `after` является последним известным идентификатором транзакции или указанным ранее курсором. Более того, потребители могут использовать готовую к использованию ссылку `before`. Поскольку ответ на этот запрос содержит только новые транзакции, его размер намного меньше. Такой вариант значительно уменьшает объем данных, скачиваемых приложением Awesome Banking, а также сокращает время

отклика, поскольку данных загружается меньше, а запросы на стороне сервера обрабатываются быстрее.

Это решение также подходит и для экрана списка транзакций по счету в маловероятном случае, когда пользователи будут прокручивать кешированные транзакции сроком более трех месяцев. В этом случае этот запрос с использованием ссылки before

```
GET /accounts/{accountId}/transactions?before=
➥ {earliestTransactionId}&size=25
```

извлекает 25 транзакций, имевших место до транзакции, определенной `earliest-TransactionId`. Предоставление параметров фильтрации – хорошая стратегия, чтобы уменьшить объем данных, которыми обмениваются, и улучшить юзабилити. Но чтобы обеспечить точные и эффективные фильтры, важно учитывать природу данных и контекст использования.

### 10.3.2 Выбор соответствующих данных для представлений списка

То, какие данные вы выбираете, чтобы вернуть их в списках, может иметь большое влияние на эффективность передачи данных. Банковский API не так эффективен, как мог бы быть, потому что он не предоставляет все релевантные данные в списках. Как показано на рис. 10.13, приложение Awesome Mobile Banking App отображает звания и имена владельцев счетов на экране панели управления.

Если имена владельцев могут быть получены только с помощью цели «Перечислить владельцев», итоговые данные не содержат званий. Чтобы получить эту информацию, приложение должно прочитать подробную информацию о каждом владельце. Это показатель неправильного баланса между итоговым представлением ресурсов, обычно возвращаемых в списках, и подробным представлением, обычно возвращаемым при обращении к конкретному ресурсу. Просто добавив звание в итоговую версию, мы можем избежать вызовов цели «Прочитать владельца».

То же самое касается цели «Перечислить счета», которая не возвращает типы счетов и, следовательно, требует дополнительных вызовов для цели «Прочитать счета», чтобы получить эту основную информацию. Добавление свойства `type` в итоговое представление, возвращаемое целью «Перечислить счета», предотвращает любые дополнительные вызовы API.

Можно было бы изменить цель «Перечислить транзакции» таким же образом, но мы пойдем еще дальше. Как показано на рис. 10.14, экран счетов должен вызывать цель «Перечислить транзакции», а затем цель «Прочитать транзакции» для каждой из них.

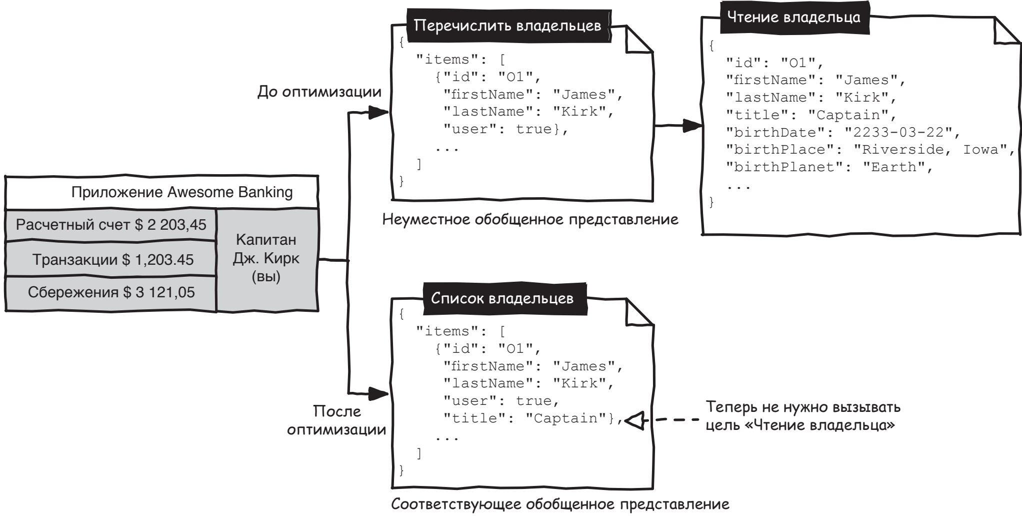



**Рис. 10.13.** Выбор соответствующих итоговых представлений в списках

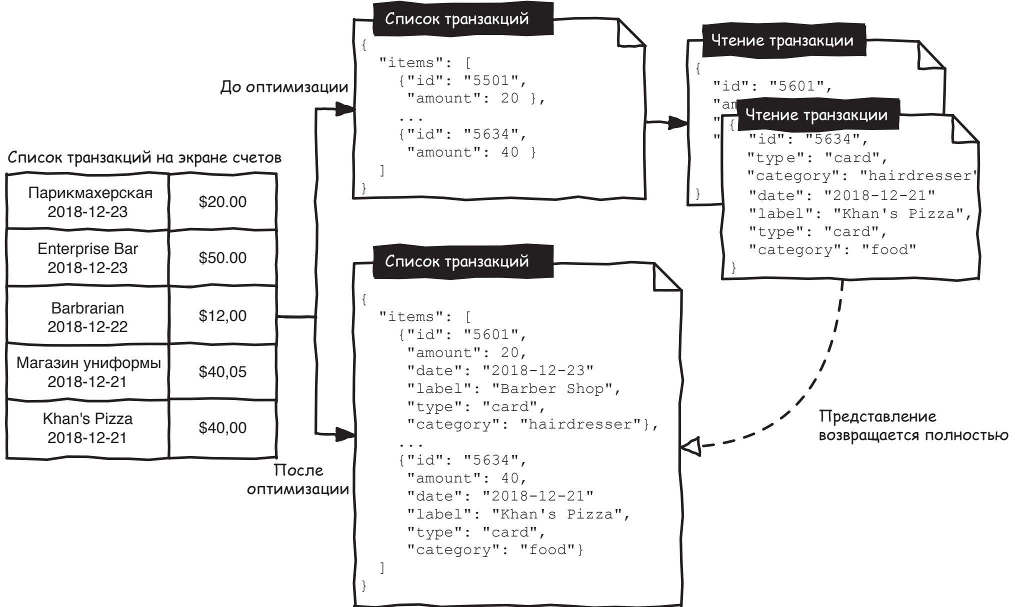



**Рис. 10.14.** Использование полных представлений в списках

Итоговое представление транзакции, возвращаемое в списке, содержит только идентификатор и метку; сумма и тип транзакции отсутствуют, поэтому потребителю нужно запрашивать подробную информацию о каждой транзакции. Учитывая характер транзакций, которые обычно многочисленны и просматриваются партиями, цель «Перечислить транзакции» должна возвращать полное представление каждой транзакции, а не просто сводку.

Почему бы не сделать такую же модификацию для других списков, таких как список владельцев? Ресурс владельца содержит гораздо больше данных, и большинство из них не имеют отношения к работе со списком. Но возвращение всех данных из списка владельца увеличит объем данных без необходимости.

Выбор подходящего представления, включающего наиболее представительные и полезные свойства ресурса, – лучший способ не только создать удобный для использования API, но и избежать большого количества вызовов API после получения данных списка. Хотя при запросе списка элементов обычно возвращается итоговая версия каждого элемента, это не является обязательным. Бывают случаи, когда возврат полного представления более эффективен.

## 10.3.3 Агрегирование данных

Детализированные ресурсы обеспечивают гибкий и точный способ получения различных подмножеств данных из концепции, но они могут привести к появлению большого количества вызовам API, когда потребителям необходимо получить все данные. Без учета предыдущих оптимизаций, которые мы сделали, на рис. 10.15 показано, как приложение Awesome Banking загружает данные о владельце.

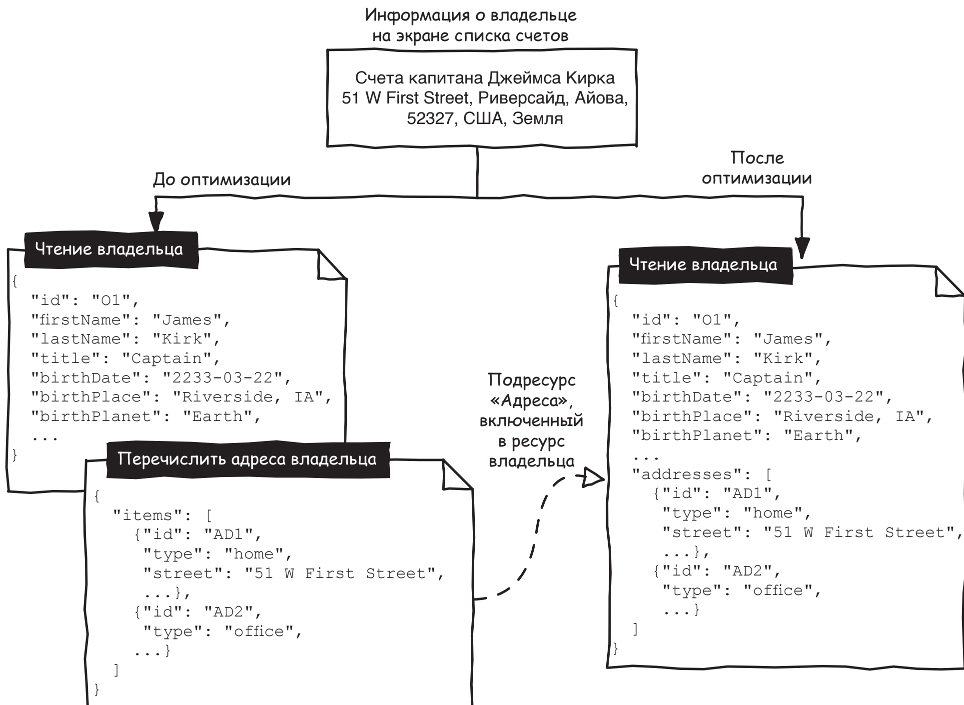



**Рис. 10.15.** Агрегирование подресурсов и родительского ресурса

Данные о владельце распределяются между ресурсом владельца, доступным с помощью цели «Прочитать владельца», и ресурсом адресов,

доступным с помощью цели «Перечислить адреса». Потребители могут получить то или иное подмножество, но это означает, что для получения двух тесно связанных и довольно небольших наборов данных необходимы два вызова API. Это неудобно; и в условиях небезопасной сети мы не можем позволить себе этот дополнительный вызов. Мы уже видели подобный вариант использования в разделе 7.2. Более подходящим вариантом было бы включить сюда список адресов с остальными данными о владельце, чтобы один вызов цели «Прочитать владельца» возвращал все необходимые данные.

Идем дальше. Почему бы нам не объединить счета и транзакции? Однако это не очень хорошая идея по нескольким причинам: по каждому счету транзакций может быть много. Потребители, возможно, захотят отфильтровать транзакции по типу или дате, и, что наиболее важно, список транзакций регулярно обновляется. Можно было бы объединить список адресов в данные о владельце, поскольку объем данных относительно невелик и адреса меняются не слишком часто. Даже если потребителям нужно выбрать адрес определенного типа, им просто нужно отфильтровать не более десяти элементов. А если данные изменятся, не нужно будет извлекать слишком много данных. Однако для транзакций по счетам лучше оставить выделенный доступ.

Таким образом, мы не можем агрегировать транзакции в данные по счету, а как насчет попытки расширенного агрегирования на другой стороне дерева? Почему бы не получить все данные, кроме списка транзакций с помощью одного вызова? На рис. 10.16 показано влияние такой агрегации для варианта, включающего пользователя, у которого есть доступ к четырем владельцам и их восьми счетам.

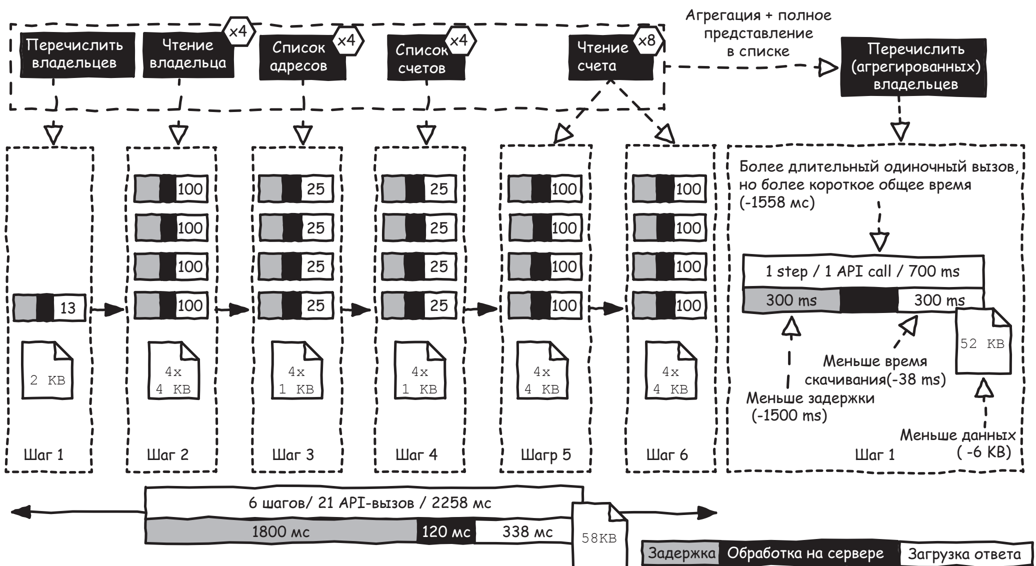



**Рис. 10.16.** Расширенная агрегация влияет на передачу данных

Извлечение всех данных о счете, кроме транзакций за один вызов, означало бы возврат полных представлений в список счетов, объединение всех этих данных в ресурс владельца вместе со списком адресов и возвращение полного представления каждого владельца в списке владельцев. Что мы получаем, заменив эти 17 API-вызовов (они осуществляются в шесть этапов, занимают 2,2 с и представляет 58 Кб данных) одним вызовом?

Время задержки сокращается с 1500 до 300 мс, поскольку вместо шести этапов используется один. Удивительно, но данных также скачивается меньше: 52 вместо 58 Кб. Это связано с тем, что дублированные данные, возвращаемые в итоговых списках, больше не скачиваются; данные возвращаются только один раз при использовании целей «Прочитать владельца» или «Прочитать счет». Время обработки сервером по-прежнему составляет 120 мс; но в действительности оно, вероятно, тоже уменьшится. Общее время уменьшается с 2,2 с до 700 мс. Теперь вместо нескольких коротких вызовов у нас есть один более длинный. Результат довольно впечатляющий; время отклика сокращается почти на 70 %!

Мы могли бы оставить эту новую цель «Перечислить агрегированных владельцев» и цель «Перечислить транзакции» и удалить остальные цели из банковского API. Но имейте в виду, что уменьшение в основном касается времени ожидания; если на сервере API активированы постоянные соединения, агрегация может быть не такой эффективной.

В определенных контекстах агрегация также может препятствовать возможностям кеширования. Время жизни агрегированных данных – это наименьшее значение из всех отдельных свойств (в данном случае это баланс счета, который может меняться довольно часто). Поэтому при изменении баланса на одном счету потребителям придется заново загружать большое количество данных. Хотя это может показаться не таким уж и важным, в условиях небезопасной сети наличие одного очень длинного вызова вместо нескольких более коротких может быть проблематично. Чем дольше длится запрос в сети 3G, тем выше риск потери соединения, а если соединение потеряно, когда загрузка была завершена на 95 %, потребителю придется скачивать все данные снова.

Наконец, помимо производительности, агрегирование может влиять на удобство использования. Потребителям может быть нелегко понять, как работает API, предоставляя только с помощью целей «Перечислить владельцев» и «Перечислить транзакции». Таким образом, агрегирование данных может быть правильным решением для решения возможных проблем, связанных с эффективностью передачи данных, но это нужно делать осторожно, хорошо представляя себе и осознавая все последствия. Проектируя ресурсы и цели, разумно выбирайте их детализацию, чтобы API был не только удобным в использовании, но и эффективным.

### *10.3.4 Предложение разных представлений*

Разумно используя агрегацию или более полные представления в списках, можно спроектировать более эффективный API. Но это довольно

жесткое решение; не всем потребителям могут понадобиться все данные во всех случаях. Как сделать наш API более адаптивным и предоставить потребителям способ выбирать представление, которое наиболее подходящим образом отвечает их потребностям?

Вы уже знаете ответ на этот вопрос: мы можем использовать согласование содержимого, возможность предоставления различных представлений ресурса, о чем вы узнали в разделе 6.2.1. Как показано на рис. 10.17, мы могли бы предоставить три разных уровня представления наших ресурсов: итоговый, полный и расширенный.

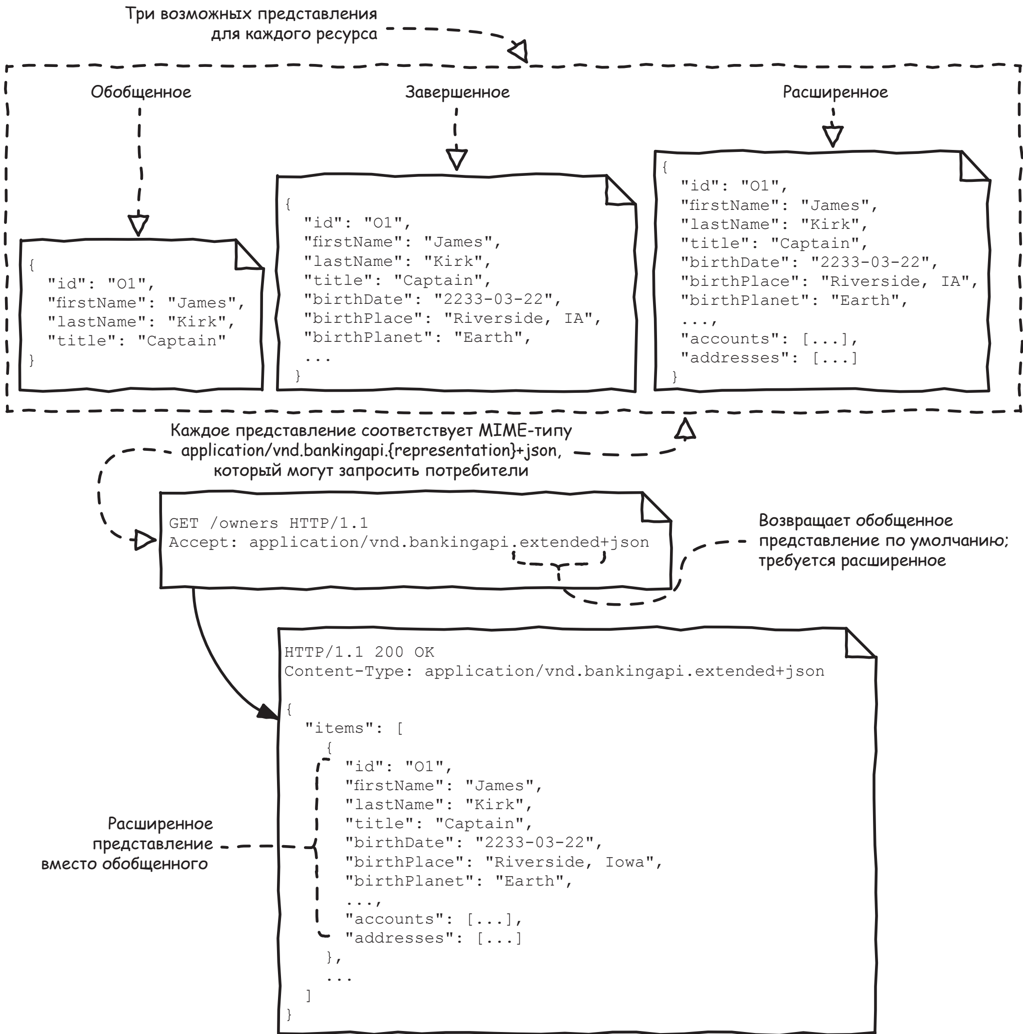



**Рис. 10.17. Использование согласования содержимого для получения соответствующего представления**

Мы привыкли получать полное представление при чтении определенного ресурса, как в случае с целью «Прочитать владельца» (`GET /owners/{ownersId}`). Итоговое представление обеспечивает подмножество дан-

ных полного представления. Это то представление, которое мы привыкли получать в списках, используя цель «Перечислить владельцев» (например, `GET /owners`). Наконец, расширенное представление представляет собой совокупность данных ресурса и его подресурсов. Здесь оно предоставляет полные данные для ресурса владельца вместе с данными его подресурсов – полные представления ресурсов счетов и адресов.

Теперь, когда приложение Awesome Banking запрашивает цель «Прочитать владельцев» на своем экране панели мониторинга, оно может указать, что ему нужно расширенное представление каждого владельца вместо итогового представления по умолчанию, отправляя заголовок `Accept`, значение которого – `application/vnd.bankingapi.extended+json`. Таким образом, он может избежать отдельных вызовов целей «Прочитать владельцев», «Перечислить счета» и «Прочитать счет».

Положительное и отрицательное влияние на скорость, кеширование и риск возникновения потерянных соединений такие же, как и те, что вы видели в разделе 10.3.3; но теперь другие потребители имеют возможность выбирать, чтобы получить только итоговое представление каждого владельца, если это все, что им нужно. Также, чтобы получить обновленный баланс счета, потребители могут отправить запрос `GET /accounts/{accountId}` вместе с заголовком `Accept: application/vnd.bankingapi.summarized+json`, чтобы получить только необходимые данные вместо обычного полного представления.

Стандартного способа обработки этого механизма не существует. MIME-типы `application/vnd.bankingapi.{representation}+json` показанные здесь, являются полностью настраиваемыми. Их имена используют стандартный префикс vnd, что значит *vendor*. Суффикс `+ json` также является стандартным и указывает, что этот пользовательский MIME-тип в основном представляет собой данные в формате JSON. Предоставление различных представлений ресурса может помочь обеспечить более эффективный и более гибкий API, но можно добиться большего.

### 10.3.5 Активация расширения

Используя согласование содержимого, можно спроектировать гораздо более гибкий API, предоставляющий, например, три разных представления владельца. Но это по-прежнему немного жестко. Что, если потребителям нужно получить только итоговые представления о владельцах наряду с их счетами, но без адресов? Это невозможно сделать, если только мы не добавим четвертое, не такое обобщенное представление владельца. Давайте попробуем кое-что еще: метод под названием *расширение ресурсов*, показанный на рис. 10.18.

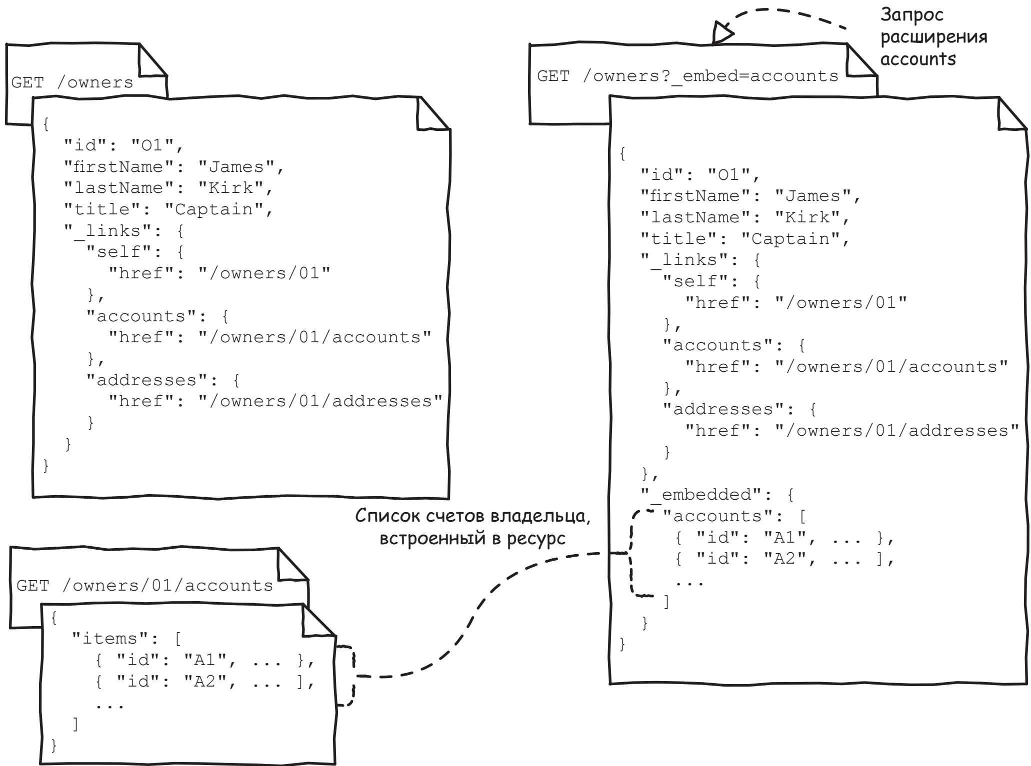



**Рис. 10.18. Расширение подресурса счетов владельца в списке владельцев**

Слева мы видим запрос на получение списка владельцев (`GET /owners`), который возвращает суммарное представление владельцев, и запрос на получение счетов (`GET /owners/01/accounts`), который возвращает итоговое представление счетов. Обратите внимание, что в этом представлении мы даем ссылки HAL в свойстве `_link` (см. раздел 6.3.2).

Справа запрос на получение списка владельцев содержит параметр запроса `_embed = accounts`, что означает «Пожалуйста, вставьте списки счетов всех владельцев в ответ». Ответ фактически включает эту информацию в свойство `_embedded.accounts`.[1]Если потребители отправляют запрос с параметром запроса `_embed=accounts`, `addresses`, возвращаемые представления владельца включают в себя списки для счетов записей и адресов в свойстве `_embedded`.

Этот параметр `_embed` позволяет нам инициировать расширение или встраивание подресурса. Опять же, здесь могут быть недостатки, включающие в себя более длинные запросы, большие ответы и неэффективность кеширования.

Стандартного способа предложения такого механизма не существует; то, что представлено здесь, полностью индивидуально. Параметр за-

[1] Это представление соответствует спецификации HAL (https://tools.ietf.org/html/draft-kelly-json-hal-06#section-4.1.2).

проса можно назвать `embed`, `expand`, или дать любое другое имя на ваш выбор. В зависимости от того, как организованы данные и какой формат гипермедиа используется (HAL, Siren, пользовательский и т. д.), способ включения подресурсов может варьироваться.

Расширение ресурсов – еще один способ уменьшить количество вызовов, которые могут потребоваться потребителям для получения дерева данных. Тем не менее дальнейшая экономия возможна посредством запросов.

### 10.3.6 Активация запросов

Если каждый байт и миллисекунда действительно имеют значение, мы можем сделать наш API еще более адаптируемым, позволяя потребителям запрашивать данные, которые им нужны, свойство за свойством, чтобы уменьшить объем данных и, возможно, количество API-вызовов. Например, запрос `GET /owners?_fields=id` может вернуть список владельцев; но для каждого владельца потребитель получит только его идентификатор. Не существует стандартного способа предложить такой механизм с помощью REST API, но обычно это делается с использованием параметра запроса под названием `fields` или `properties` (или чего-то подобного), значение которого представляет собой список имен свойств (например, `title`) или путей JSON (например, `$ .accounts[*].id`, чтобы получить все идентификаторы счетов).

В качестве альтернативы, если требуются более сложные запросы, чтобы уменьшить объем данных, вы можете рассмотреть еще один вариант: существующий язык запросов. REST не единственный способ создания API. Мы уже кратко говорили о gRPC, но существует еще один стиль API, который может представлять интерес: GraphQL. GraphQL, язык с открытым исходным кодом, созданный Facebook в 2012 году, – это

> «*Язык запросов для API и среда выполнения для осуществления этих запросов с вашими существующими данными*».
>
> https://graphql.org

Этот раздел не предназначен, для того чтобы научить вас создавать API-интерфейсы GraphQL; он предназначен только для того, чтобы предоставить пример существующего языка API-запросов, который вы могли бы использовать, чтобы потребители могли запрашивать нужные им данные, вместо того чтобы создавать свои собственные. В приведенном ниже листинге показан базовый вызов GraphQL, который запрашивает список владельцев. Это эквивалентно запросу `GET /owners?fields=id`.

**Листинг 10.1. Вызов GraphQL API и его ответ**

```
POST /graphql
{
  "query": "{ owners { id } }"
}
HTTP/1.1 200 OK
```

```
{
  "owners": [
    {"id": "01"},
    {"id": "02"},
    ...
  ]
}
```

Вызов GraphQL API состоит из запроса `POST /graphql`. Его тело – это документ в формате JSON, который при чтении данных содержит свойство строки `query`. Значением этого свойства является фактический запрос GraphQL, который будет выполняться для извлечения данных.

Не обманывайтесь фигурными скобками; этот запрос написан не в формате JSON! Запрос `{owners {id}}` гласит, что нам нужен только идентификатор каждого владельца. В приведенном ниже листинге показан более длинный запрос GraphQL, который входит в свойство query и получает дополнительные данные о владельцах и их счетах.

**Листинг 10.2. Получение данных о владельце и счете**

```
{
  owners {
    id
    title
    firstName
    lastName
    accounts {
      id
      balance
    }
  }
}
```

Этот запрос возвращает список владельцев, содержащий выбранные данные. Чтобы сделать это с помощью REST Banking API (без предоставления агрегированных данных), нам нужно было бы объединить несколько вызовов API. Сначала мы перечислили бы владельцев с помощью запроса `GET /owners`, а затем перечислили бы счета каждого владельца с помощью запроса `GET /owners/{ownerId}/accounts`.

Теперь представьте, что банковский API предлагает цель, позволяющую нам получить список ближайших банкоматов. При использовании REST API мы будем использовать такой запрос:

```
GET /atms?latitude=48&longitude=2&distance=2
```

чтобы получить список банкоматов, находящихся в двух милях от указанного места. Но, как показано в приведенном ниже листинге, мы можем выполнить два запроса, используя один вызов API GraphQL, чтобы получить список владельцев и их счетов, а также список ближайших банкоматов.

**Листинг 10.3. Выполнение нескольких запросов**

```
{
  owners {
    id
    title
    firstName
    lastName
    accounts {
      id
      balance
    }
  }
  atms (latitude: 48, longitude: 2, distance: 2) {
    address
    longitude
    latitude
  }
}
```

Потребители могут легко выбрать именно те данные, которые они хотят, и сделать несколько запросов за один вызов. Но, поскольку GraphQL использует только HTTP-метод POST, запросы нельзя кешировать с использованием стандартного механизма кеширования HTTP, тогда как запрос `GET /atms?latitude=48&longitude=2&distance=2` – можно.

На момент написания этой книги GraphQL не предлагал никакого механизма кеширования; потребители должны догадаться, как долго они смогут кешировать данные. Как и в случае с агрегацией данных, кеширование ответа в целом может не иметь смысла, поскольку оно может содержать разнородные данные с очень разными значениями времени жизни. Есть и другие последствия, которые необходимо оценить, прежде чем выбрать такое решение; мы немного поговорим об этом в разделе 11.3.1.

Активирование запросов данных может быть целесообразным в некоторых сценариях, но не во всех. Это может уменьшить объем передаваемых данных и количество API-вызовов, но за счет возможностей кеширования.

### 10.3.7 Предоставление более релевантных данных и целей

Как вы только что видели, приложение Awesome Banking может получать все данные, необходимые для любого из его экранов, за один вызов с использованием языка запросов API. Но, прежде чем рассмотреть вопрос об изменении типа банковского API с REST на GraphQL, мы должны пересмотреть его дизайн. Неэффективный обмен данными может быть признаком дизайна, который не удовлетворяет фактическим потребностям потребителей.

В разделах 10.3.2 и 10.3.3 мы уже видели, что наш выбор в отношении степени детализации ресурсов и того, какие данные мы включаем

в сводные представления, влияет не только на эффективность передачи данных, но и, что более важно, на удобство использования. Но для обеспечения дизайна, который был бы удобен для использования и эффективен, требуется больше, чем просто выбор того, какие данные возвращать в списки и как распределять ресурсы, – предоставление *релевантных* данных и целей является ключом к созданию такого дизайна.

Когда потребители должны получить информацию о счете, им обычно нужны его тип, имя, баланс и история транзакций. Банковский API обеспечивает все это, благодаря целям «Прочитать счет» и «Перечислить транзакции». Но баланс счета изменяется каждый раз, когда происходит новая транзакция.

Используя текущий дизайн, обновление этой информации может быть выполнено путем запроса последних транзакций с использованием пагинации на основе курсора (см. раздел 10.3.1) или с помощью условного запроса (см. раздел 10.2.2). Если появляются новые транзакции, потребители должны снова прочитать счет, чтобы получить обновленный баланс, хотя все остальные данные счетов, вероятно, не изменились вообще. Потребители банковского API, вероятно, никогда не будут использовать список транзакций без остатка на счете.

Это, безусловно, тесно связанные данные: баланс основан на суммах транзакций. Как показано в приведенном ниже листинге, добавление обновленного баланса счета к каждой транзакции может упростить ситуацию.

**Листинг 10.4. Добавление обновленного баланса в транзакции**

```
{
  "items": [
    {"id": "5601", "date": "2018-12-23", "amount": 20,
    ➥ "balance": 202.3, ...},
    {"id": "5550", "date": "2018-12-23", "amount": 20,
    ➥ "balance": 222.3, ...},
    {"id": "5548", "date": "2018-12-22", "amount": 23.7,
    ➥ balance": 246, ...},
    ...
  ]
}
```

При получении новых транзакций потребители теперь будут автоматически получать обновленный баланс счета каждый раз без необходимости его повторного чтения. В качестве бонуса эта модификация предоставляет интересную информацию: потребители могут увидеть, как баланс изменился с течением времени. Обратите внимание: не потому, что баланс счета был добавлен к транзакциям, его следует удалить из цели «Прочитать счет»; баланс полезен в обоих местах.

Предоставление релевантных данных также означает не предоставление всех доступных данных. Если сосредоточиться на точке зрения потребителя, это может помочь ограничить объемы данных (помните раз-

дел 2.4.1?). В нашем случае ресурсы владельца и счетов банковского API, возможно, могли бы пропустить несколько неинтересных свойств, которые имеют значение только для реализации поставщика API.

> **ПРИМЕЧАНИЕ.** Добавление нужных данных к нужным ресурсам и ориентация на точку зрения потребителя с целью предоставления только тех данных, которые действительно актуальны, могут повысить как удобство использования, так и эффективность использования сети.

В корне древовидной структуры ресурсов банковского API находится список доступных владельцев; все потребители должны пройти через этот корень, чтобы сделать что-нибудь. Это кажется подходящим для приложения Awesome Banking, на экранах которого отображаются данные с той же структурой, что и API. Но это означает, что все потребители должны перечислить владельцев с помощью вызова `GET /accounts/{ownerId}/accounts`, чтобы узнать, какие счета доступны. Это может раздражать тех, кому не особо интересны владельцы счетов. Но когда консультанты хотят получить обзор всех счетов всех своих клиентов, было бы полезно добавить в API запрос `GET /accounts`, с помощью которого можно было вернуть все счета, к которым могут получить доступ текущие пользователи (кем бы они ни были).

Кроме того, среди владельцев, возвращаемых целью «Перечислить владельцев», один соответствует конечному пользователю. В текущем варианте потребители, желающие получить данные только о конечном пользователе, должны перечислить владельцев и найти в возвращаемом списке того, чей флаг `endUser` имеет значение `true`. Используя магический идентификатор ресурса, такой как `me`, потребители могли бы напрямую читать информацию о конечном пользователе, используя цель «Прочитать владельца» с запросом `GET /owners/me`, без необходимости сначала перечислять владельцев для определения идентификатора пользователя.

Как видите, добавление дополнительных целей, обеспечивающих разный доступ к одним и тем же ресурсам или более прямой доступ, также может повысить удобство использования и эффективность в разных контекстах. Например, способ, которым приложение Awesome Banking создает свой основной экран панели инструментов, может привести к добавлению данных к существующим целям или даже к созданию более конкретных целей. Объединение сумм транзакций и остатков на счетах по владельцам может выполняться реализацией API и добавляться к данным владельца. Объединение сумм транзакций также можно добавить к данным, возвращаемым целями «Прочитать счет» и «Перечислить транзакции» наряду с балансом счета.

Если это имеет смысл для других потребителей, мы также могли бы рассмотреть возможность добавить цель «Прочитать панель управления», доступную через запрос `GET /dashboards/me`, которая будет возвращать данные, необходимые для экрана панели управления приложения Awesome Banking. Если из такой модификации могут извлечь выгоду многие потребители, ее следует добавить в API.

Также имейте в виду, что потребители, вероятно, будут использовать ваши API неожиданным образом. Будь то вопиющие дыры в первоначальном варианте или из-за того, что у некоторых потребителей просто есть идеи, о которых вы никогда и не мечтали, разумно проанализировать такие неожиданные варианты использования и изменять конструкцию по мере необходимости, чтобы обеспечить наиболее эффективный интерфейс. Для проектировщиков API очень важно оценить эффективность своей работы.

Как вы видели в разделе 10.1.2, в зависимости от варианта использования эффективность банковского API сильно варьируется: для загрузки данных панели управления может потребоваться пять вызовов API, завершающихся за 1,2 с, или 25 вызовов, завершающихся за 4,2 с. Оценивая эффективность передачи данных, не нужно думать только о базовых сценариях использования. Потоки целей API могут выглядеть идеально при очень простом гипотетическом сценарии использования и превращаться в кошмары, когда вы сталкиваетесь с реальностью или крайними случаями. Проектировщики API всегда должны тестировать свои проекты в реальных условиях, чтобы по-настоящему оценить их эффективность.

### *10.3.8 Создание разных слоев API*

Попытка оптимизировать API с целью эффективности передачи данных по сети – это хорошо, но Проектировщики API должны знать, когда следует сказать «нет». Оптимизация дизайна API для обеспечения эффективной передачи данных не должна осуществляться за счет удобства использования и возможности повторного использования. Попытка угодить всем потребителям путем внесения определенных изменений здесь или там или добавления нескольких весьма специфических целей, вероятно, приведет к появлению сложного API, который нельзя будет использовать повторно. К счастью, используя различные методы, описанные в этой главе, вы должны быть в состоянии спроектировать эффективный API, и это должно дать вам уверенность в необходимости дать отпор при необходимости.

Если у потребителей действительно есть специфические потребности, им следует создавать свои собственные API-интерфейсы поверх API поставщиков. В мире мобильных приложений и сайтов такой компонент называется BFF (это не *best friends forever*, а *backend for frontend*). Команда проектировщиков приложения Awesome Banking могла бы, например, создать компонент BFF на базе GraphQL, опираясь на банковский API. Это довольно просто; существуют библиотеки GraphQL, которые могут помочь разработчикам сделать это без необходимости написания кода.

Поставщики также могут сами предоставлять такие API, создавая новый слой API в своих системах. Такие специализированные API иногда называют *Experience API* (независимо от их типа – REST, GraphQL и т. д.). Их дизайн оптимизирован для конкретного контекста использования с функциональной или технической (обычно сетевой) точки зрения.

Ниже этого слоя можно найти *оригинальные/неспециализированные API*. Это API-интерфейсы, дизайн которых ориентирован на потребителя, но на самом деле они не ограничены конкретным контекстом использования. А ниже этого слоя вы можете найти *системные API*, предоставляющие доступ к основным системам. Если вы помните пример с микроволновой печью из раздела 2.1, такой API предоставляет доступ к магнетрону.

В следующей главе мы полностью изучим контекст, окружающий API, как со стороны потребителя, так и со стороны поставщика, чтобы разработать API, которые были бы более удобными для использования и реализуемыми.

### Резюме

- Проектировщики API играют свою роль в эффективности передачи данных по сети.
- Самый первый этап в оптимизации сети находится на уровне протокола, а не на уровне проектирования.
- Детализация и адаптируемость API оказывают влияние на эффективность сети.
- Проблемы с эффективностью сети могут быть признаком недостающих или неадекватных целей в API.
- Оптимизация дизайна API не должна выполняться за счет юзабилити и возможности повторного использования; предоставление различных слоев API может помочь избежать таких ловушек.

# 11 Проектирование API в контексте

**В этой главе мы рассмотрим:**

- адаптацию обмена данными к целям и данным;
- учет потребностей и ограничений потребителей и поставщиков;
- выбор стиля API на основе контекста.

В предыдущей главе мы начали понимать, что проектируемые нами API создавались без учета большей части контекста, в котором они существуют. Мы изучили сетевой контекст и его влияние на дизайн API. Но есть и другие контекстные элементы, которые необходимо учитывать для проектировании API, удовлетворяющего потребности всех ваших клиентов, а также реализуемого. Как мы уже видели, проектирование API требует, чтобы мы в первую очередь фокусировались на потребителях, но и чтобы следили за стороной поставщика.

Знаете ли вы, что раскладка клавиатуры QWERTY была изобретена в конце XIX-го века? Наиболее распространена история, согласно которой она была создана для решения механической проблемы. На пишущей машинке буквы прикреплены к металлическим кронштейнам, которые могут столкнуться и заклинить, если две соседние клавиши нажаты одновременно или в быстрой последовательности. Чтобы избежать такой механической проблемы и дать пользователям печатать быстрее, часто используемые пары букв располагались далеко друг от друга. Эта история, если это правда, означает, что на дизайн QWERTY повлияли

внутренние проблемы. Но, по словам Коичи Ясуока и Мотоко Ясуока из Киотского университета,[1]

> *«Первая клавиатура для пишущей машинки была взята из телетайпа Хьюза и Фелпса. Она разрабатывалась для приемников Морзе. Расположение клавиатуры очень часто менялось в процессе разработки и случайно превратилось в QWERTY. QWERTY был принят телетайпом в 10-х годах XX века, а телетайп позже широко использовался в качестве компьютерного терминала».*
>
> *Коичи Ясуока и Мотоко Ясуока*

Согласно этому исследованию, на дизайн фактически влиял контекст, в котором использовались пишущие машинки. Независимо от своего происхождения забавно то, что этот реликт из прошлого по-прежнему широко используется и сегодня. Я осмотрел свой смартфон и не обнаружил никаких металлических рычагов за сенсорным экраном; но в большинстве стран латинские или римские алфавитные цифровые клавиатуры все еще используют раскладку QWERTY или их локальную версию, например AZERTY во Франции. Даже если это больше не имеет смысла, люди привыкли к этому, и те немногие, кто осмелился попытаться изменить свои привычки, не добились успеха. Таким образом, как объекты создаются и как они работают, как они используются и к чему привыкли их пользователи, могут влиять на их дизайн, и то же самое касается и API, как показано на рис. 11.1.

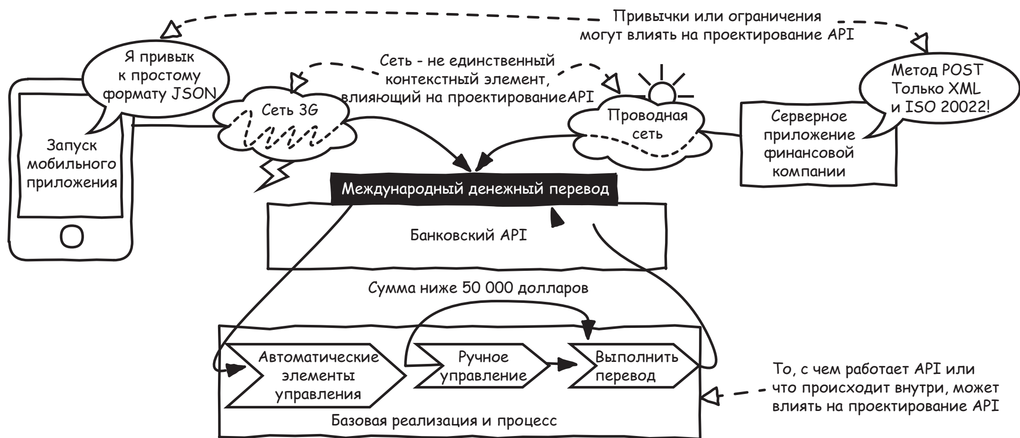



**Рис. 11.1.** Контексты поставщика и потребителя влияют на проектирование API

Хотя большинство разработчиков, возможно, привыкли к использованию API на базе JSON, используя все преимущества протокола HTTP, существуют темные уголки индустрии программного обеспечения, где по-прежнему правит XML, а `POST` – это единственный возможный HTTP-метод. Банковская индустрия привыкла к сообщениям стандарта ISO 20022, ко-

[1] *Коичи Ясуока* и *Мотоко Ясуока*, «О предыстории QWERTY», Киотский университет, март 2011 г (https://doi.org/10.14989/139379).

торые можно рассматривать как сложные и не удобные для пользователя, но попытка предоставить банковским компаниям API, поддерживающие другие более простые форматы, может вызвать больше проблем, чем те, которые эти форматы призваны решать.

Дизайн, влияющий на контекст, также не зарезервирован для потребительского контекста. Контекст поставщика также может влиять на дизайн, даже если проектировщики API делают все возможное, чтобы скрыть точку зрения поставщика (см. раздел 2.4). Представление цели с участием человеческого контроля (например, в некоторых случаях международных денежных переводов) с механизмом синхронного запроса/ответа, возможно, не лучший вариант. Вот почему, проектируя API, мы должны выбрать наиболее подходящий способ передачи данных, принимая во внимание потенциальные ограничения как потребителей, так и поставщиков и даже учитывая другие стили API, помимо REST. Если мы этого не сделаем, API-интерфейсы, которые мы проектируем , могут оказаться не полностью пригодными для использования или реализуемыми.

## 11.1 Адаптация передачи данных к целям и характеру данных

До сих пор мы говорили о синхронных веб-API, которые позволяют потребителям отправлять запросы поставщикам и незамедлительно получать ответы. Но в зависимости от характера целей и данных API унитарный и механизм синхронного запроса/ответа, возможно, не самое эффективное представление. Возможно, вам придется иметь дело с длительным временем обработки, отправлять события потребителям или обрабатывать несколько элементов за один раз. Будучи проектировщиком API, вы должны располагать в своем наборе инструментами, отличными от синхронного запроса и ответа, инструментами, чтобы справляться с такими случаями.

### 11.1.1 Управление длительными процессами

Механизм синхронного запроса/ответа не всегда является лучшим вариантом для представления цели. Иногда вам может потребоваться предоставить асинхронные цели. Например, банковский API предоставляет цель «Перевести деньги», которая позволяет осуществлять как национальные, так и международные переводы. Но в соответствии с банковскими правилами, в зависимости от того, в какой стране и в каком банке находится целевой счет и какова сумма перевода, возможно, потребуется предоставить некоторые документы, чтобы объяснить характер транзакции. Таким образом, потребитель (например, приложение Awesome Banking) должен предоставить исходный счет и счет назначения для каждого перевода (см. раздел 5.3). Чтобы определить их, он использует цель «Перечислить исходные счета и счета назначения».

Данные, возвращаемые этой целью, не только описывают все возможные комбинации исходного счета и счета назначения и их минимальные и максимальные суммы. В них также указано, в каких случаях должна

быть предоставлена документация, оправдывающая транзакцию. Если требуется документация, потребитель может использовать цель «Загрузить документ перевода», чтобы отправить его и получить ссылку. На рис. 11.2 показано, что происходит потом: проверка человеком.

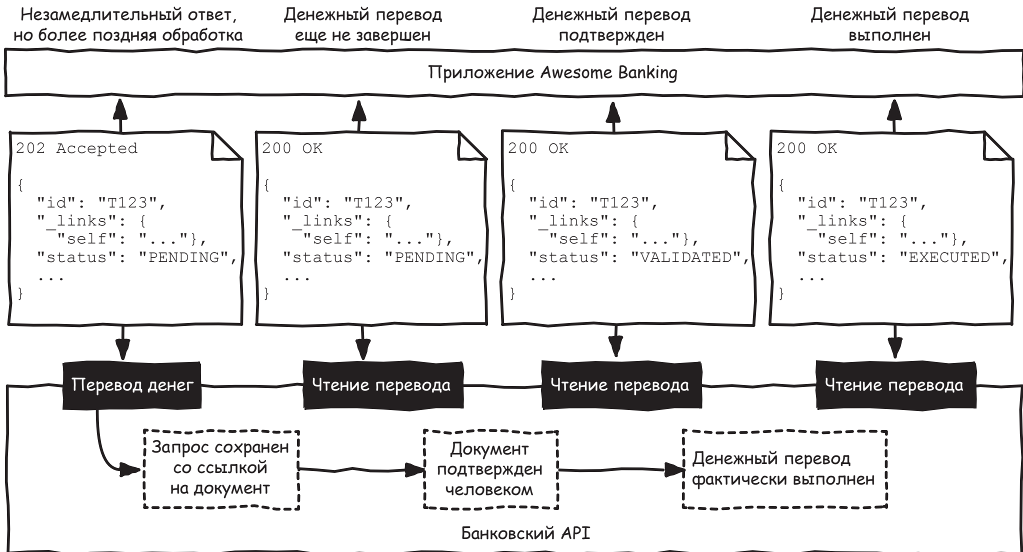



**Рис. 11.2.** Денежный перевод, требующий проверки со стороны человека

Как только документ будет загружен, потребитель сможет использовать цель «Перевести деньги», указав исходный счет, счет назначения, сумму и ссылку на документ. К сожалению, перевод денег нельзя инициировать тотчас же, потому что предоставленный документ должен быть подтвержден человеком. Поэтому в этом случае код состояния – `202 Accepted` (вместо `201 Created`, который появляется в качестве ответа, если проверка не требуется). Это означает, что запрос на перевод денег был принят, но будет обработан позже.

Возвращаемые данные указывают на текущий статус перевода (`PENDING`), идентификатор перевода (`T123`) и URL-адрес «`self`» в _links. Позже потребитель может использовать цель «Прочитать перевод» и предоставленный идентификатор или собственный URL-адрес, чтобы проверить текущий статус перевода. Это может быть либо `PENDING` (если еще не выполнено никакого действия), `VALIDATED` (если документ был подтвержден человеком, но перевод еще не был выполнен) или `EXECUTED` (если перевод был завершен). Обратите внимание, что при обращении к статусу перевода с помощью HTTP-запроса `GET /transfers/T123` директивы кеша (см. раздел 10.2.2) могут дать подсказки о том, когда имеет смысл повторить этот вызов, чтобы получить обновленную информацию.

Как видите, в зависимости от характера цели то, что на самом деле происходит с функциональной точки зрения с использованием механизма синхронного запроса/ответа, может оказаться невозможным. Здесь это будет означать, что потребители ждут несколько минут (или даже часов,

если не дней), чтобы получить ответ, что, очевидно, немыслимо. В таких случаях API должен предоставить цель для получения запроса, что может занять достаточно много времени для обработки, а затем способ получить статус обработки этого запроса позже. Предоставление информации о том, когда сделать еще один запрос, используя преимущества протокола или просто возвращая данные, выгодно как потребителю, так и поставщику. При этом не нужно делать ненужных вызовов.

### 11.1.2 Уведомление потребителей о событиях

Связь между потребителем и поставщиком не всегда является наиболее эффективным способом общения. Иногда бывает полезно позволить поставщику проявить инициативу. В предыдущем разделе мы увидели, что потребителям, возможно, придется повторять вызовы API, чтобы спросить: «Этот денежный перевод осуществлен?» Такое поведение называется опросом, и оно может быть довольно раздражающим как для потребителей, так и для поставщиков: можно сделать много ненужных вызовов. Было бы здорово, если бы вместо этого банковский API мог сообщить своим потребителям, когда денежный перевод все-таки был сделан.

Обратный обмен данными между потребителем и провайдером можно выполнить с помощью веб-хука, который часто называют *обратным API*. На рис. 11.3 показано, как можно использовать такой механизм с приложением Awesome Banking для уведомления потребителя о совершенном денежном переводе.

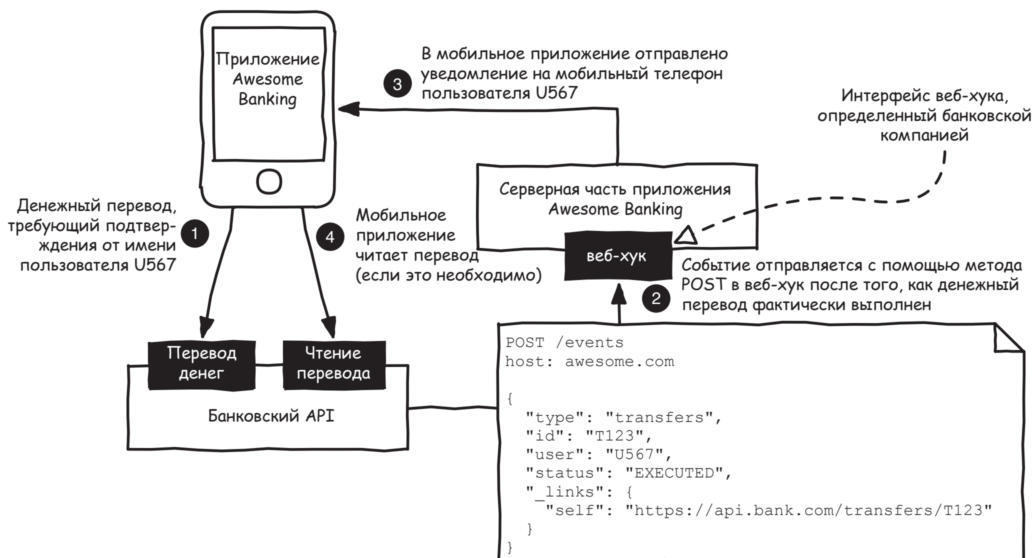



**Рис. 11.3.** Использование веб-хука для уведомления потребителя о выполнении денежного перевода

Как и прежде, приложение Awesome Banking вызывает банковский API для запроса на перевод денег, который требует проверки (со сторо-

ны человека) (1). Банковский API снова отвечает кодом состояния `202 Accepted`, чтобы указать на то, что запрос был принят и будет обработан позже. Теперь мобильное приложение не должно опрашивать (регулярно совершать вызовы) банковский API для получения статуса перевода. Вместо этого, когда денежный перевод фактически будет выполнен, банковский API (или, вероятнее всего, еще один модуль в системах банковской компании) отправляет запрос `POST` в URL-адрес веб-хука приложения Awesome Banking, https://awesome-banking.com/events (2). Тело запроса содержит некие данные о произошедшем событии, например, идентификатор пользователя, инициировавшего перевод денег, идентификатор перевода, статус события и ссылку перевода «self».

Когда серверная часть приложения Awesome Banking, реализующая веб-хук, получает это событие, она может найти идентификатор мобильного телефона, соответствующий пользователю, и отправить уведомление с помощью системы уведомлений iOS или Android в мобильное приложение, чтобы сообщить, что денежный перевод `T123` выполнен (3). Наконец, мобильное приложение может использовать цель «Прочитать перевод», чтобы получить дополнительную информацию, которая не была включена в событие или уведомление (4).

Такой механизм не ограничивается асинхронной передачей данных, инициированной потребителем. Его также можно использовать для уведомления потребителей о событиях, которые генерируются без какого-либо взаимодействия с потребителем. Например, события могут отправляться, когда происходят новые транзакции. Приложение Awesome Banking может использовать это для экранов панели управления, владельцев и счетов (см. раздел 10.2), а также может использовать кешированные данные, если такие события не отправляются.

Также можно отправлять более конкретные и пользовательские события. Например, банковский API может предоставить систему оповещения, которая отправляет события на основе транзакций или данных о балансе. Используя такую функцию, приложение Awesome Banking может позволить своим пользователям настраивать такие оповещения, как «Дайте знать, когда остаток на моем счету будет ниже 200 долл.» или «Дайте знать, когда будет произведена оплата картой на сумму свыше 120 долл.» Банковский API будет отправлять эти события оповещения через веб-хук только тем пользователям, которые их настроили.

Все это выглядит великолепно, но откуда банковская компания, поставщик банковского API, знает URL-адрес веб-хука приложения Awesome Banking и его интерфейс? В разделе 8.1 вы видели, что потребители должны зарегистрироваться, чтобы иметь возможность использовать API, и что они идентифицируются при отправке запроса. При регистрации приложения Awesome Banking на портале разработчиков банковского API команда разработчиков указала URL-адрес своего веб-хука, который можно использовать для уведомления потребителя о событиях.

Этот веб-хук – API, реализованный командой разработчиков приложения Awesome Banking, но его контракт интерфейса и поведение определяются командой банковского API, чтобы гарантировать, что все по-

требители предоставляют доступ к одному и тому же веб-хуку. Очевидно, что для банковской компании было бы кошмаром позволить каждому потребителю проектировать собственный контракт, поскольку ей пришлось бы писать код для вызовов веб-хука для всех своих потребителей.

Как и в случае с любыми API-интерфейсами, вы должны проектировать веб-хуки, чтобы скрыть точку зрения поставщика и сделать их удобными для использования и способными к развитию. В зависимости от ваших потребностей один веб-хук может получать все возможные события или веб-хуков может быть несколько: по одному на каждый тип событий. Каждое событие может предоставить небольшое количество данных или много данных. Вам решать, что уместно.

Наличие единственного веб-хука, который получает упрощенные, типовые события, обычно является хорошей стратегией. Такой API довольно прост в реализации и использовании, и добавлять новые события будет легко. Всегда нужно решать, какой дизайн использовать в соответствии с вашим контекстом.

Есть еще одна важная характеристика, которую нельзя игнорировать при работе с веб-хуками – это безопасность. К веб-хуку можно получить доступ через интернет, и некоторые злоумышленники могут попытаться отправить *ложные* события, чтобы взломать системы поставщика. Это одна из причин, по которой использование упрощенных типовых событий является хорошим вариантом; потребители должны связаться с поставщиком, чтобы получить подробную информацию.

Крайне важно, чтобы доступ к веб-хуку был защищен, чтобы гарантировать, что только поставщик API может использовать его. Защиту веб-хука можно осуществлять с использованием различных методов, таких как внесение в белый список IP-адресов провайдера (имейте в виду, что такие белые списки может быть трудно поддерживать), отправка секретного токена при регистрации события в веб-хуке, шифрование и подпись запроса с помощью взаимной аутентификации с использованием протокола TLS и т. д.

Как вы видели в главе 8, проектировщикам API особо нечего рассказать о технической стороне безопасности API, но они вносят большой вклад с функциональной точки зрения. Вы должны убедиться, что события не содержат конфиденциальных данных и что предоставленные данные позволяют потребителям безопасно реагировать. Например, если событие касается конкретного пользователя, потребители API должны иметь возможность идентифицировать этого пользователя по данным события. В противном случае пользователь может получить лишний доступ к данным других пользователей. Это способствует использованию упрощенных событий для ограничения ущерба, который может быть нанесен, если это произойдет.

### WebSub

Стандарта для веб-хуков не существует. Хотя вы можете создать свой API по своему усмотрению, консорциум Всемирной паутины W3C выпустил рекомендацию по WebSub (https://www.w3.org/TR/websub/), которой вы можете воспользоваться при создании систем на базе веб-хуков:

> *«WebSub предоставляет общепринятый механизм для обмена данными между издателями веб-контента любого типа и их подписчиками на базе веб-хуков с использованием протокола HTTP. Запросы на подписку передаются через хабы, которые проверяют и подтверждают запрос. Затем хабы распространяют новый и обновленный контент подписчикам, когда он становится доступным. Ранее WebSub был известен как PubSubHubbub».*
>
> *Рекомендация по WebSub от W3C*

В основном в этой рекомендации описывается, как поставщик API (издатель) может позволить потребителям (подписчикам) регистрироваться и получать события безопасным способом. Вдохновленный этой рекомендацией, банковский API может предоставить стандартный API, позволяющий потребителям регистрироваться для получения таких событий, как механизм оповещений, рассмотренный ранее в этом разделе.

Веб-хуки в основном представляют собой API, реализованные потребителями API, но определенные и используемые поставщиками API для отправки уведомлений о событиях. Эти события могут быть инициированы действиями потребителя или поставщика. Это не единственный способ реализации уведомлений, но с помощью этой модели поставщики могут уведомлять потребителей о событиях, когда они происходят, и не нужно ждать, пока потребители сами выполнят API-вызовы.

### 11.1.3 Потоковая передача событий

Когда API предоставляет данные, которые всегда изменяются на потребителей, используя основную цель запрос/ответ, вы можете быть уверены, что они будут непрерывно опрашивать их, делая повторные вызовы API для получения новых или обновленных данных. Предположим, что банковский API предоставил данные об акциях для портфелей торговых счетов. Для этого есть разные варианты, как показано на рис. 11.4.

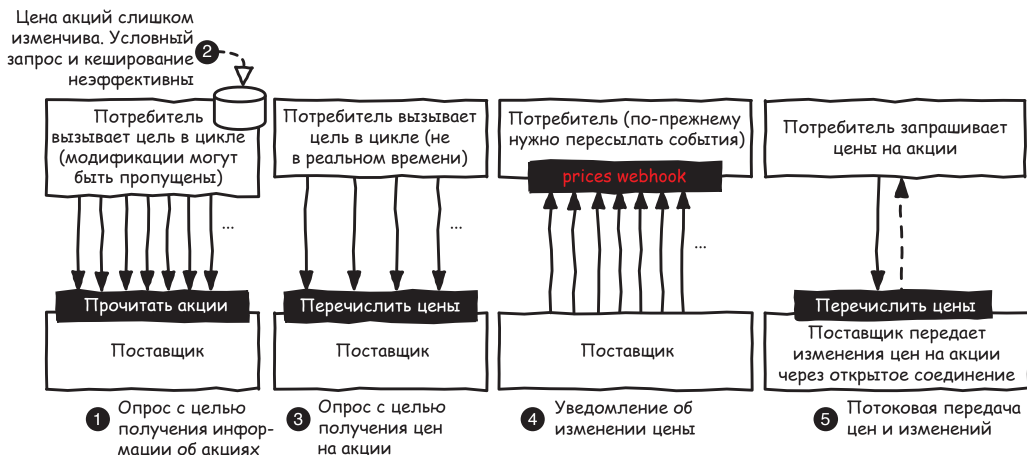



**Рис. 11.4.** **Как банковский API может предоставлять информацию об акциях**

Банковский API может предложить цель «Прочитать акции», которая предоставляет подробную информацию о конкретной акции и ее стоимости (1). Потребители, желающие всегда получать самые последние цены на акции, могут вызвать эту цель в цикле (как только они получают новые данные, они инициируют еще один вызов, и так до бесконечности).

В разделе 10.2.2 вы познакомились с кешированием и условными запросами (2). Могут ли они помочь? К сожалению, они бесполезны, по крайней мере, когда фондовые биржи открыты, потому что цены на акции могут меняться каждую секунду (если не чаще).

Время жизни кеша будет настолько коротким, что кеширование будет неэффективным, а условные запросы всегда будут возвращать обновленные данные. Эти данные настолько изменчивы, что даже при использовании опроса потребители могут и не знать обо всех колебаниях цен, поскольку цена может варьироваться между вызовами.

В разделе 10.3.7 вы видели, что иногда нужно проверять, действительно ли цель удовлетворяет потребности потребителей. Может быть, эта не та цель, которая нужна. Как насчет того, чтобы адаптировать дизайн API и предоставить цель «Перечислить цены на акции», которая возвращает *n*-е число последних изменений цен и предлагает пагинацию на базе курсора (3)? Потребители могут указать идентификатор последний цены, которую они получили и, таким образом, они будут уверены, что не пропустили ни одного изменения в цене. Этот вариант может быть интересным, если потребители хотят получать данные не совсем в реальном времени, но они все равно будут бесконечно опрашивать эту цель.

Изменение цены на акции выглядит как событие, о котором потребители могут быть уведомлены. Так что насчет веб-хука (4)? Поскольку поставщик знает, когда цена акций меняется, он может регистрировать это событие в веб-хуках потребителей, как только оно происходит.

Но это значит всегда рассылать изменения цен на все акции всем потребителям.

Используя систему WebSub или пользовательскую систему, подобную WebSub, как мы кратко обсуждали в предыдущем разделе, потребители могли бы подписаться на несколько каналов изменения цен на акции вместо получения уведомлений обо всех. Они все равно будут получать эти данные все время.

Но как быть с потребителями, которые хотят показывать данные в режиме реального времени только в течение небольшого периода времени, в то время как их конечные пользователи смотрят, например, на свои портфели или конкретные акции? Им придется найти способ пересылки этих потоков событий. Как сервер API (например, сервер банковского API) может отправлять поток событий, запрошенных потребителем (5)? На рис. 11.5 приводится сравнение базового API- вызова запрос/ответ и потока SSE, который можно использовать в таких случаях.

**Рис. 11.5.** Потоковая передача событий потребителям с помощью технологии SSE

В верхней части рисунка потребитель запрашивает последние цены на акции APL в виде документа `application/json`, используя запрос `GET /stocks/APL/prices` с соответствующим заголовком Accept. По умолчанию сервер возвращает список цен за последние пять минут. В документе есть атрибут `items`, содержащий список цен. Чтобы получить более свежие данные, потребитель должен будет сделать еще один запрос, используя пагинацию на базе курсора.

В нижней части рисунка показано, как все это можно обработать с помощью потока SSE. Запрос почти такой же, но потребитель теперь указывает, что ему нужны данные в виде документа `text/event-stream`. Сервер отвечает кодом состояния `200 OK` и документом с заголовком `Content-Type: text/event-stream`, как и было запрошено. Каждое ценовое событие представлено строкой, начинающейся с `data:`, и содержит те же данные, что и ранее, в готовом списке.

Огромная разница в этом подходе состоит в том, что теперь ценовые события предоставляются в виде потока; возвращаемый документ больше не является статичным и законченным. Сервер добавляет строку с data: для каждого нового ценового события, происходящего для акций APL. Это будет продолжаться, до тех пор пока больше событий не будет или пока потребитель не закроет соединение. Используя SSE, сервер может отправлять данные о событиях потребителям.

Что касается данных о событиях, вы, вероятно, помните, что в случае использования веб-хука я рекомендовал помещать в события как можно меньший объем данных и чтобы потребители получали дополнительные данные с помощью очередного регулярного вызова API. Но в данном случае, независимо от используемой технологии (SSE или какой-либо еще), обычно лучше предоставить как можно больше данных, поскольку потребители захотят получить все данные без необходимости повторного независимого вызова API. Однако, как и всегда, это не обязательно; все может зависеть от контекста. Также здесь применимо и все остальное, о чем вы узнали из этой книги: события и их данные должны иметь смысл для потребителей и должны быть простыми для понимания и использования, чтобы развиваться и обеспечивать безопасность.

Обратите внимание, что использование согласования содержимого и предоставление MIME-типов `application/json` и `text/event-stream` не требуется; банковский API может предоставлять только потоковую версию. Также необязательно использовать один и тот же путь для предоставления двух этих разных представлений. Банковский API может использовать разные пути, такие как `/stocks/{stockId}/prices` и `/stocks/{stockId}/price-events`. API также может предоставить способ получения ценовых событий для нескольких акций с помощью запроса типа `GET /stock-prices?stockIds=APL,APA,CTA`. В ответ на этот запрос каждое событие, отправленное с помощью строки с `data:` будет касаться одной из акций APL, APA или CTA; потребитель сможет определить, какой из них, проверив значение свойства `stockId`.

Хотя спецификация SSE предоставляет больше возможностей, чем просто строки с data:, она довольно проста. Приведенный ниже листинг показывает различные возможности.

**Листинг 11.1 Полная спецификация SSE[1]**

: this is a comment ①

[1] «W3C Working Draft», апрель 2009 г., ред. Ян Хиксон, Google, Inc. (https://www.w3.org/TR/2009/WD-eventsource-20090421/#event-stream-interpretation).

```
data: this is text data ②
③
data: {«json»: «data»} ④

data: this is multi- ⑤
data: line data

id: optional event ID ⑥
event: optional event type
data: event data

retry: 10000 ⑦
```

① **Поток можно прокомментировать строкой, начинающейся со знака двоеточия (:).**
② **Каждая строка, начинающаяся со слова data, – это событие; в основном она содержит текст.**
③ **Пустая строка отделяет каждое событие.**
④ **Поскольку данные - это текст, можно использовать JSON или XML.**
⑤ **Несколько строк с data: можно использовать для многострочных данных.**
⑥ **Каждое событие может быть дополнено дополнительным идентификатором и типом события.**
⑦ **Интервал повторных попыток дает потребителю указание не подключаться, до того как пройдет 10000 мс (10 с), если соединение потеряно.**

Есть еще несколько вещей, которые нужно знать о SSE:

- она использует протокол HTTP, но не является его частью; она была создана W3C как стандарт для HTML5;
- ее довольно просто использовать для потребителей на базе браузера, потому что она была разработана для них, но есть библиотеки, доступные практически для любого языка;
- данные события могут быть только текстовыми (простой текст, JSON, XML и т. д.). Если вам нужно отправить двоичные данные, например изображения, необходимо закодировать их в текст;
- поток SSE может использовать сжатие HTTP. Это однонаправленный поток, который означает, что после установки соединения потребитель не может отправлять данные на сервер, использующий это соединение.

Поскольку эта технология использует протокол HTTP, для размещения API, использующего ее, не требуется никакой конкретной инфраструктуры, но имейте в виду: использование SSE означает, что соединения HTTP остаются открытыми в течение достаточно долгого времени. Поэтому инфраструктура, в которой размещается API, должна быть настроена на поддержку длинных параллельных соединений. Тем не менее было бы полезно использовать единый поток SSE для отправки различных типов событий. Для этого можно воспользоваться свойством event.

Теперь предположим, что банковская компания хочет предоставить функции чата, позволяющие конечным пользователям обсуждать свои счета с людьми или ботами. В этом случае возможно было бы предпоч-

тительнее обеспечить двунаправленную передачу данных, позволяющую потребителю и поставщику отправлять события. К сожалению, SEE допускает использование только однонаправленной передачи данных от сервера к потребителю; но, к счастью, есть и другие решения. Такая потребность обычно удовлетворяется с помощью протокола WebSocket, как определено в RFC 6455 (https://tools.ietf.org/html/rfc6455), который широко применяется в чатах и играх. Мы не будем вдаваться в подробности инфраструктуры, но знайте, что этот подход требует больше работы на стороне инфраструктуры, чем поток SSE на базе протокола HTTP.

WebSocket использует необработанное TCP-соединение, которому, возможно, запрещено проходить через корпоративные прокси-серверы без изменения их конфигурации. Что касается сообщений, которыми можно обмениваться по этому протоколу, то вы, как проектировщик API, должны выполнять свою работу, не полагаясь на какие-либо стандарты. Но помните, что вы можете копировать то, что уже делали другие.

Большинство WebSocket API используют типизированные сообщения, как и в SSE, за исключением того, что в этом случае и потребитель, и поставщик могут отправлять сообщения. Если вам нужно связать запрос на событие, отправленный через WebSocket, с реакцией на него, вам просто нужно добавить в сообщения какой-нибудь уникальный идентификатор.

**ПРИМЕЧАНИЕ.** WebSocket также можно использовать для однонаправленной передачи данных, как и в случае использования SSE.

Существуют разные способы потоковой передачи событий. Для разработчика API важно знать, что механизм запроса/ответа не единственный вариант. При работе с данными с высокой волатильностью и данными в реальном времени потоковая передача событий не только от поставщика к потребителю, но и от потребителя к поставщику является вариантом, который следует учитывать.

### *11.1.4 Обработка нескольких элементов*

Различные примеры API, которые вы видели до сих пор, предоставили два способа чтения данных. Некоторые цели могут обеспечить доступ к отдельным элементам, а другие – к нескольким.

Например, банковский API позволяет потребителям читать один счет с помощью цели «Прочитать счет» или несколько счетов с помощью цели «Перечислить счета». Но когда дело доходит до целей, связанных с созданием, изменением или удалением, мы видим только те цели, которые работают с одним элементом. В зависимости от элементов, которыми манипулируют, и контекста возможно полезно иметь возможность обрабатывать несколько элементов с помощью одного вызова API, вместо того чтобы выполнять множество вызовов, каждый из которых обрабатывает один элемент за раз.

Чтобы изучить эту тему, давайте добавим еще несколько функций управления личными финансами в банковский API. Мы можем позволить потребителям изменять транзакции для определения персонализированных категорий, добавлять комментарии о них, а также проверять

их; проверка транзакции похожа на пометку письма как прочитанного. Для этого мы добавим цель «Обновить транзакцию», представленную запросом `PATCH /transactions/{actionId}`. В приведенном ниже листинге показана JSON-схема ожидаемого тела.

**Листинг 11.2. JSON-схема тела цели «Обновить транзакцию»**

```
openapi: "3.0.0"
...
components:
  schemas:
    ...
    UpdateTransactionRequest:
      description: |
        At least one of the comment, customCategory, or
        ➥ checked properties must be provided
      properties: ①
        comment:
          type: string
          example: My new Ibanez electric guitar
        customCategory:
          type: string
          example: Music Gear
        checked:
          type: boolean
          description: |
            Checking a transaction is similar to
            ➥ marking an email as read.
            ➥ True if the transaction has been
            ➥ checked, false otherwise.
```

① **Тело состоит из трех свойств: comment, customCategory и checked.**

Свойства `comment`, `customCategory` и `checked` являются необязательными; потребители могут обновить одно из них, два или все три. Идентификатор транзакции также не требуется, поскольку он указывается в пути к ресурсу `/transactions/{actionId}`. В этом случае никакие другие свойства транзакции, такие как ее сумма или дата, не могут быть обновлены.

Предоставление функции «Пометить все как отмеченные» или «Пометить все выбранные как прочитанные» в приложении Awesome Banking требует от нас обновления каждой транзакции. В зависимости от количества транзакций при вызове API для обновления каждой транзакции может возникнуть проблема. Как мы видели при чтении данных (раздел 10.3), можно было бы попытаться объединить все эти единичные вызовы в один. То есть мы могли бы разрешить потребителям обновлять несколько транзакций в одном API-вызове, предлагая для этого цель «Обновить транзакции» (где слово «транзакция» стоит во множественном числе). Как показано на рис. 11.6, такую цель мож-

но обозначить запросом PATCH /transactions. В листинге 11.3 показана JSON-схема его тела.

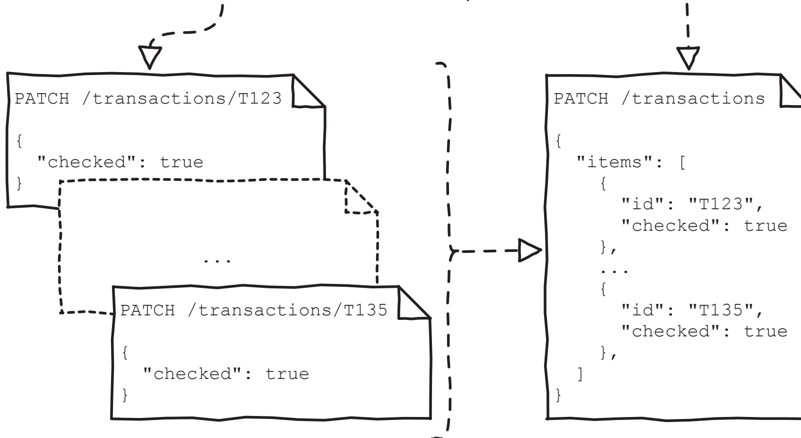



**Рис. 11.6.** Проверка нескольких транзакций в одном вызове

**Листинг 11.3. JSON-схема тела цели «Обновить транзакции»**

```
openapi: "3.0.0"
...
components:
  schemas:
    ...
    UpdateTransactionsRequest:
      properties:
        required:
          - items
        items:
          type: array
            minItems: 1
            maxItems: 100 ①
            items:
              allOf: ②
                - required:
                    - id
                properties:
                  id:
                    type: string
                    ) description: Transaction ID
                - $ref: "#/components/schemas/
                ➥ UpdateTransactionRequest"
```

① Не более 100 обновлений одновременно.

② Те же данные, что и единичный вызов плюс идентификатор транзакции; агрегаты allOf предоставляют JSON-схемы.

Обновленные данные для всех транзакций предоставляются в виде объекта, содержащего свойство `items`, которое представляет собой список от 1 до 100 транзакций. Это тот же тип представления, который используется в теле ответа цели «Перечислить *что-либо*», например «Перечислить транзакции». Свойства, предоставляемые для каждой транзакции, аналогичны свойствам, указанным для единичной цели (`comment`, `customCategory` и `checked`) плюс идентификатор, поскольку идентификатор транзакции не может находиться в пути к ресурсу. Чтобы проверить несколько транзакций за один вызов, потребители должны предоставить свойства `id` и `checked` для каждой проверенной транзакции.

Это что касается запроса, а как насчет ответа? При обновлении одной транзакции цель «Обновить транзакцию» может означать, что обновление было выполнено с кодом состояния `200 OK`, что-то не так с запросом с кодом состояния `400 Bad Request` или что идентификатор транзакции неизвестен – `404 Not Found`. При обработке нескольких транзакций одновременно, если все транзакции успешно обновлены, может вернуться код состояния `200 OK`.

То же самое происходит в случае ошибки: ошибка `400 Bad Request` может возвращаться, даже если проблема заключается в неверном идентификаторе транзакции. Код состояния `404` может быть возвращен только в том случае, если путь к ресурсу неизвестен, что в данном случае не так. Это немного отличается от единичного обновления. А что, если какие-то транзакции можно обновить, а какие-то нет? Должна ли реализация банковского API остановиться на первой ошибке и вернуть код состояния `400` без обработки каких-либо допустимых обновлений транзакций?

Если вы помните наше обсуждение в разделе 5.2.4, то знаете, что ответ на этот вопрос – «нет», потому что это сделает API менее пригодным удобным для использования. Потребителям пришлось бы сделать большое количество вызовов, чтобы исправить все ошибки одну за другой (и API, не обрабатывающий действительные обновления, может привести в бешенство). Цель «Обновить транзакции» должна возвращать все ошибки, обрабатывать все действительные обновления транзакций, а также указывать, какие обновления были выполнены успешно. Это означает возврат нескольких статусов; к счастью, для этого есть HTTP-код:

> *«Код состояния 207 (мультистатус) предоставляет статус для нескольких независимых операций…»*
>
> *WebDAV*

Статус `207` определен в спецификации RFC 4918, что позволяет клиентам выполнять удаленные операции по авторизации веб-контента.[1] Он предоставляет новые методы, заголовки, MIME-типы и статусы, чтобы облегчить управление ресурсами и в особенности управление несколькими ресурсами за один вызов. В приведенном ниже листинге показан

[1] «HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)», L. Dusseault, Ed., Июнь 2007 (https://tools.ietf.org/html/rfc4918).

пример того, что должен возвращать сервер WebDAV, согласно RFC 4918, в ответе с кодом 207 при удалении нескольких ресурсов.

**Листинг 11.4. Ответ 207 Multi-Status, как описано в спецификации RFC 4918**

```
<?xml version="1.0" encoding="utf-8" ?>
<d:multistatus xmlns:d="DAV:">
   <d:response>
      <d:href>http://www.example.com/container/resource3
      ➥ </d:href>
      <d:status>HTTP/1.1 423 Locked</d:status>
      <d:error><d:lock-token-submitted/></d:error>
   </d:response>
   <d:response>
      <d:href>http://www.example.com/container/resource4
      ➥ </d:href>
      <d:status>HTTP/1.1 200 OK</d:status>
   </d:response>
</d:multistatus>
```

Это XML-документ, содержащий список, каждый элемент которого состоит из `href` (URL-адреса обрабатываемого ресурса), `status` (единичный статус HTTP) и необязательного сообщения об ошибке. Обратите внимание на сжатие и кодирование на верхнем уровне (ответ, отправленный сервером API); каждый ответ должен использовать одну и ту же кодировку и сжатие.

Спецификация RFC 4918 описывает различные ответы с кодом `207` для определенных методов HTTP (таких как `PROPPATCH` и `PROPFIND`), но они основаны на XML – слишком ориентированы на контекст WebDAV, – и их нельзя использовать повторно в других контекстах. Вот почему я решил оставить статус `207` и определить собственный формат (JSON) для тел запросов и ответов для цели «Обновить транзакции», как показано на рис. 11.7.

Ответ `207 Multi-Status` на запрос обновить транзакции – это объект со свойством items, который представляет собой список, содержащий столько же элементов, сколько и в списке `items`, предоставленном в запросе. Список ответов упорядочен точно так же, как и список запросов: ответ на третий запрос находится на третьей позиции в списке ответов. По каждому элементу потребители получают точно такую же информацию, которую они получили бы, совершая единичный вызов. Здесь это означает `status` и body, содержащие статус HTTP и тело ответа для каждой попытки обновления транзакции.

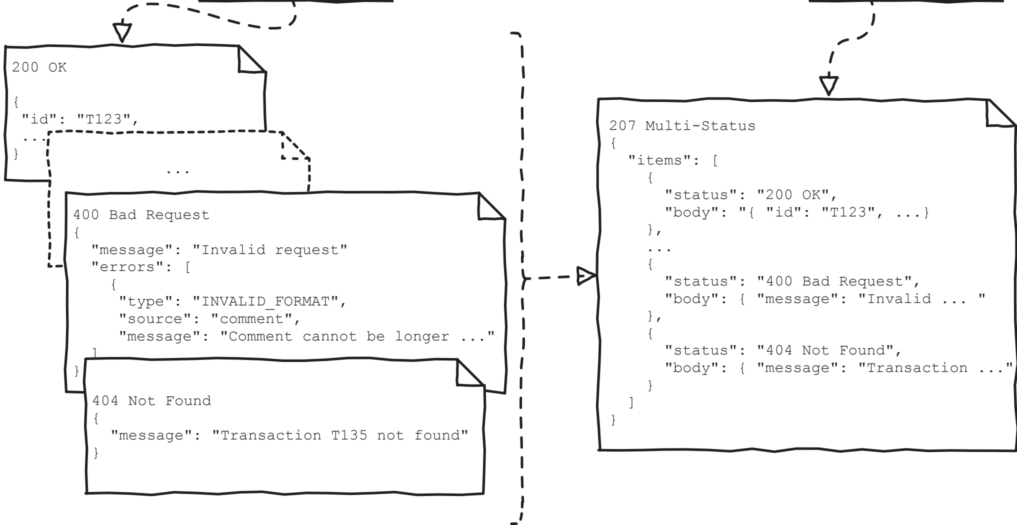



**Рис. 11.7.** Сравнение нескольких ответов на цель «Обновить транзакцию» с одним ответом на цель «Обновить транзакции»

Первый статус в списке – статус, свидетельствующий об успехе (200 OK), а его тело содержит обновленный ресурс. Последние два запроса не были обработаны из-за слишком длинного комментария и неизвестного идентификатора транзакции. Для каждого из них body содержит структуру данных с ошибками (см. раздел 5.2.4), которая была бы возвращена для единичного вызова. Если потребитель отправляет неверный список в своем запросе (например, с более чем 100 элементов), код состояния будет 400 Bad Request. Также, если заголовки обычно возвращаются для единичных вызовов, можно было бы добавить ассоциативный массив headers для каждого элемента. В листинге 11.5 показана полная схема JSON, а в листинге 11.6 приводится пример.

**Листинг 11.5.** Схема JSON ответа с несколькими состояниями

```
openapi: "3.0.0"
...
components:
  schemas:
    MultipleStatusResponse:
      required:
        - items
      properties:
        items: ①
          type: array
          minItems: 1
        maxItems: 100
        items:
```

```
        required:
          - status
        properties:
          status: ②
            type: string
            description: HTTP status
            example: 404 Not Found
          headers:
            additionalProperties: ③
              type: string
            description: HTTP headers map
            example:
              My-Custom-Header: CUSTOM_VALUE
              Another-Custom-Header:
              ➥ ANOTHER_CUSTOM_VALUE
          body:
            description: |
              Transaction if status is 200 OK,
              ➥ Error otherwise
            oneOf: ④
              - $ref: "#/components/schemas/Error"
              - $ref: "#/components/schemas/
              ➥ Transaction"
            example:
              message: Transaction T135 not found
```

① Содержит один элемент для каждой транзакции запроса.
② Статус HTTP.
③ Ассоциативный массив <string, string> для заголовков (это также может быть список имен, значений).
④ – Тело – Транзакция, если статус 200 OK или Ошибка в противном случае (одна из предоставленных схем JSON).

**Листинг 11.6. Пример, сгенерированный с использованием JSON-схемы**

```
{
  "items": [
    {
      "status": "404 Not Found",
      "headers": {
        "My-Custom-Header": "CUSTOM_VALUE",
        "Another-Custom-Header": "ANOTHER_CUSTOM_VALUE"
      },
      "body": {
        "message": "Transaction T135 not found" }
    }
  ]
}
```

Помните, что то, что показано на рис. 11.7 и в листингах 11.5 и 11.6, является моей собственной интерпретацией того, как может выглядеть содержимое ответа `207 Multi-Status`. Мы можем использовать один и тот же вариант для замены или удаления нескольких ресурсов с помощью запросов `PUT /resources` или `DELETE /resources?ids=1,2,5,6,9`. Чтобы создать несколько ресурсов одновременно, нужно учесть ряд вещей.

Мы могли бы использовать запрос `POST /resources`, но что, если мы также хотим, чтобы потребители могли создавать один ресурс за раз? Пока цель «Создать ресурсы» может создать один или несколько ресурсов, потребитель может передать один ресурс в списке. Мы также могли бы принять список ресурсов и один ресурс в теле запроса (вы должны попытаться описать такую операцию, ее тело запроса и различные ответы, используя спецификацию OpenAPI, как описано в главе 4). Но что, если нам нужно сделать четкое разделение по соображениям безопасности, например между целями «Создать ресурс» и «Создать ресурсы», с использованием разных путей?

Нередко встречаются запросы `POST /resources/batch` для создания нескольких ресурсов за один вызов; такие пути нарушают шаблон `/collection/{resourceId}`, но, по крайней мере, потребители поймут с первого взгляда, что они могут сделать. В зависимости от того, как выполняется защита, предоставление двух разных путей может быть неизбежным. В идеальном мире, однако, я бы предпочел предоставить один путь `POST /resources`, принимающий список ресурсов, или один ресурс потребителям, у которых группе «Создание ресурса пакета» разрешено отправлять только запросы, содержащие список ресурсов.

Имейте в виду, что стратегия частичной обработки (обработки допустимых элементов, даже если предоставленный список содержит недействительные), обсуждаемая в этом разделе, возможно, не единственная, которую следует выбирать во всех случаях. В некоторых случаях обработка только части предоставленных элементов может вызвать проблемы. Поэтому, прежде чем вводить такое поведение, всегда проверяйте последствия подобной частичной обработки. Если частичная обработка не имеет смысла, API может вернуть более классические коды состояния `200 OK` в случае успеха и `400 Bad Request`, например если запрос недействителен.

Как видите, API не обязаны предоставлять способы обработки только отдельных ресурсов; существуют контексты, в которых обработка нескольких ресурсов за один вызов может быть полезной. Независимо от того, какое решение вы проектируете, помните, что потребители должны получать одни и те же данные, включая данные протокола, такие как заголовки или коды состояния, и ошибки, которые они получили бы при единичных запросах. Они должны иметь возможность устанавливать связь между каждым элементом своего запроса и каждым элементом ответа API. И не забывайте обрабатывать глобальные элементы управления и ошибки; например, ограничивая количество элементов, которые могут быть предоставлены в запросе.

## 11.2 Соблюдение полного контекста

В разделе 10.1 вы видели, что проектирование API требует, чтобы мы думали о том, как API будут использоваться потребителями, в основном на благо потребителей, но также и на благо поставщика. И теперь мы обнаружили, что это также требует от нас заботы об истинном характере целей или данных, чтобы обеспечить эффективные, удобные в использовании и реализуемые API (см. раздел 11.1). Все это означает, что проектирование API требует от нас большего, чем просто сосредоточиться на потребностях потребителей и избегать точки зрения поставщика. Проектирование API требует от нас полного соблюдения контекста, в котором они будут использоваться и предоставляться, чтобы гарантировать, что они наилучшим образом удовлетворяют все нужды потребителей – и фактически реализуются поставщиками.

### 11.2.1 Знание существующих практик и ограничений потребителей

Удовлетворение всех нужд потребителей означает проектирование API-интерфейсов, которые обеспечивают все необходимые цели в понятном и простом для использования виде; это также означает, что нужно быть осторожным с некоторыми аспектами, которые можно назвать *нефункциональными требованиями*. Эти требования в основном касаются того, как на самом деле будут представлены цели и данные API. Потребители, возможно, привыкли к определенным методам или имеют некоторые ограничения, которые необходимо учитывать при проектировании API.

В разделе 5.1 и в разделах 6.1.3 и 6.1.4 вы видели, что API-интерфейсы, спроектированные с использованием простых представлений и стандартов, следуя общепринятым методам, легче понять, их проще использовать и они более функционально совместимы. Но это может выходить далеко за рамки простого использования кристально ясных имен и стандартных форматов дат или применения часто используемых шаблонов путей. Существующие практики могут оказать более глубокое влияние на проектирование API.

Например, предположим, что банковская компания хочет предоставить API для проверки банковских реквизитов, который подтверждает, существует ли номер счета в каком-либо банке и принадлежит ли он определенному лицу. Такой сервис может быть полезен для компаний, использующих прямое дебетование для платежей. Чтобы получить оплату, компании снимают средства с банковского счета своего клиента. При этом компании будут рады иметь уверенность в том, что предоставленная информация фактически соответствует существующему банковскому счету, принадлежащему их клиенту, прежде чем продавать ему какие-либо товары или продукты.

Такой API кажется довольно простым в проектировании. Он предлагает одну цель для проверки банковских реквизитов. Для этой цели требуется номер счета в формате IBAN, а также имя и фамилия владельца счета. В случае успеха возвращается простой ответ `OK` и предоставляется подробная информация при ошибке, например если номер счета суще-

ствует, но имя владельца не совсем совпадает из-за опечатки. Основываясь на том, что вы уже изучили, как бы вы представили такую цель? На рис. 11.8 показаны три способа сделать это.

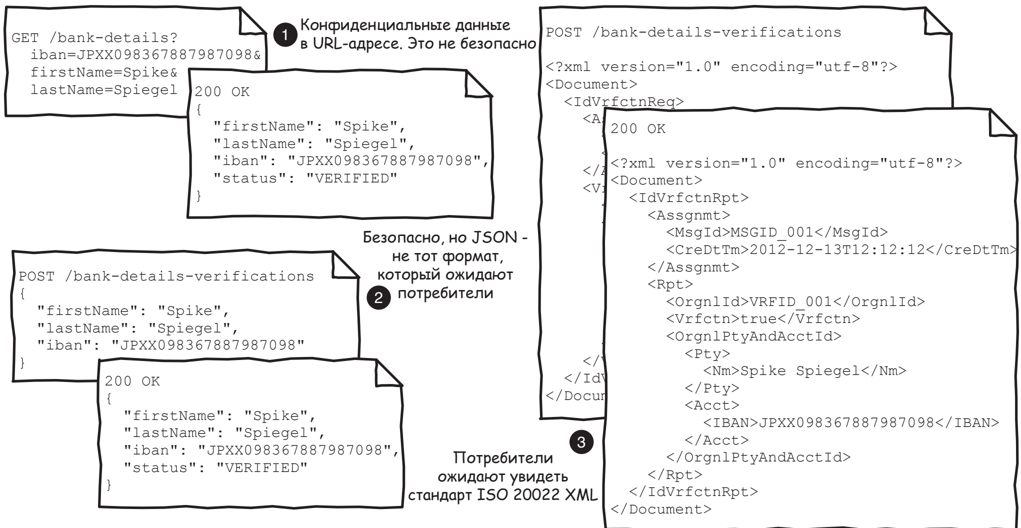



**Рис. 11.8.** Адаптация дизайна к тому, к чему привыкли потребители

Основываясь на том, что вы изучили в разделе 8.4, вы знаете, что не стоит представлять цель для проверки банковских реквизитов с помощью запроса `GET /bank-details` с параметрами запроса `firstName`, `lastName` и `iban` (1). IBAN (номера счетов) и имена и фамилии в качестве конфиденциальных данных нельзя передавать как параметры запроса – они могут быть зарегистрированы где угодно! Таким образом, вы, вероятно, представляете эту цель с помощью запроса `POST /bank-details-validation` (2), тело которого представляет собой JSON-объект, содержащий обязательные свойства `iban`, `firstName` и `lastName`. Если запрос является действительным, он может вернуть ответ `200 OK`, тело которого содержит статус подтверждения, указывая на то, действительны ли предоставленные банковские реквизиты или нет. Если запрос недействителен (например, если был предоставлен IBAN с неверным форматом или свойство `lastName` не было предоставлено), может быть возвращен ответ `400 Bad Request`, содержащий JSON- объект с информативным сообщением и сведениями о возникшей проблеме (проблемах), как вы видели в разделе 5.2.4.

Такой API кажется простым для понимания и его легко может использовать кто угодно. Но перед проектированием этого API мы не проверяли контекст реальных потребителей; и в этом случае это критическая ошибка. Целевые потребители – это корпоративные потребители банковской компании, которые будут работать с API, используя готовое к коммерческому применению финансовое программное обеспечение. Люди, работающие с таким программным обеспечением (и само программное обеспечение), не привыкли к пользовательским данным

в формате JSON; скорее, они привыкли к формату сообщений ISO 20022 XML (3). Давайте внимательнее посмотрим на этот третий вариант: и запрос, и ответ основаны на формате сообщений ISO 20022 XML, как показано в листингах 11.7 и 11.8.

**Листинг 11.7. XML-сообщение ISO 20022 IdentificationVerificationRequestV02**

```
<?xml version="1.0" encoding="utf-8"?>
<Document>
   <IdVrfctnReq>
      <Assgnmt>
         <MsgId>MSGID_001</MsgId>
         <CreDtTm>2012-12-13T12:12:12</CreDtTm>
      </Assgnmt>
      <Vrfctn>
         <Id>VRFID_001</Id>
         <PtyAndAcctId>
            <Pty>
               <Nm>Spike Spiegel</Nm> ①
            </Pty>
            <Acct>
               <IBAN>JPXX098367887987098</IBAN> ②
            </Acct>
         </PtyAndAcctId>
      </Vrfctn>
   </IdVrfctnReq>
</Document>
```

① **Имя и фамилия владельца счета.**
② **IBAN счета.**

**Листинг 11.8. XML-сообщение ISO 20022 IdentificationVerificationReportV02**

```
<?xml version="1.0" encoding="utf-8"?>
<Document>
   <IdVrfctnRpt>
      <Assgnmt>
         <MsgId>MSGID_001</MsgId>
         <CreDtTm>2012-12-13T12:12:12</CreDtTm>
      </Assgnmt>
      <Rpt>
         <OrgnlId>VRFID_001</OrgnlId>
         <Vrfctn>true</Vrfctn> ①
         <OrgnlPtyAndAcctId> ②
            <Pty>
               <Nm>Spike Spiegel</Nm>
            </Pty>
```

```
            <Acct>
               <IBAN>JPXX098367887987098</IBAN>
            </Acct>
         </OrgnlPtyAndAcctId>
      </Rpt>
   </IdVrfctnRpt>
</Document>
```

① **Статус проверки.**

② **Проверенная информация (имя и IBAN).**

Сообщение IdentificationVerificationRequestV02, показанное в листинге 11.7, представляет собой стандартный запрос для проверки банковских реквизитов. Он содержит IBAN в свойстве Document.IdVrfctnReq.Vrfctn.PtyAndAcctId.Acct.IBAN, а также имя и фамилию владельца в свойстве Document.Vrfctn.PtyAndAcctId.Pty.Nm. Есть также идентификатор запроса и дата и идентификатор подтверждения.

Сообщение IdentificationVerificationReportV02, приведенное в листинге 11.8, – стандартный ответ на такой запрос. Оно содержит исходные данные запроса и логический флаг, который имеет значение true, если проверка прошла успешно, или false в противном случае (Document.IdVrfctn-Rpt.Rpt.Vrfctn).

Поскольку стандарт ISO 20022 описывает только сообщения, а не то, как они передаются, мы можем, по крайней мере, сохранить дух второй версии API. Цель проверки банковских реквизитов по-прежнему можно представить запросом POST /bank-details-verifications, но теперь ее телом является XML-сообщение IdentificationVerificationRequestV02. Может быть возвращен ответ 200 OK, если исходное сообщение IdentificationVerificationRequestV02 является допустимым, но теперь тело ответа – это сообщение IdentificationVerification-ReportV02. Если запрос недействителен, возвращается ответ 400 Bad Request вместе с пользовательским XML-сообщением, отображенным в сообщение об ошибке в формате JSON, к которому мы привыкли (стандарт ISO 20022 не описывает, как должны обрабатываться такие ошибки).

Получившийся в итоге API не так уж и плох, но в соответствии с тем, что вы узнали, особенно в разделе 5.1, такие сообщения могут считаться сложными (и также можно было бы учитывать, что формат XML уже не в моде). Но в этом контексте целевые потребители изначально общаются с использованием сообщений в формате ISO 20022 XML; и поэтому API должен их использовать. Работать с XML API в готовом к коммерческому применению финансовом программном обеспечении, используемом целевыми потребителями, будет довольно просто. Если API применяет пользовательские сообщения в формате JSON, потребление API может потребовать дополнительной работы на стороне потребителя, а в некоторых случаях может оказаться невозможным. Но в чужой монастырь со своим уставом не ходят.

Выбор подходящего представления – это не выбор того, к чему мы, проектировщики API, привыкли, или того, что можно было бы считать

хорошим или модным решением; речь идет о выборе того, что подходит для желаемого контекста. Всегда проверяйте, если целевая область или потребители применяют конкретные практики, которым вы должны следовать при проектировании API. К таким практикам относится использование стандартов или способ представления данных, именования объектов, управления ошибками или что-либо еще.

Это учитывает существующие методы, которые могут повлиять на проектирование API, а что насчет ограничений? Допустим, что банковская компания также хочет ориентироваться на некорпоративных/нефинансовых потребителей, которые определенно не привыкли к формату ISO 20022 XML, и им больше по душе простой JSON. Умный API может использовать согласование содержимого (см. раздел 6.2.1) для решения этой проблемы. Потребители должны будут просто задать для заголовков `Content` и `Accept` значение `application/xml`, если они хотят использовать стандарт ISO 20022 и `application/json` для использования простого формата JSON. Замечательно – API достаточно адаптируем, чтобы удовлетворить потребности двух разных типов потребителей.

Но к сожалению, после опроса ряда разработчиков финансовых систем COTS, используемых целевыми клиентами, создается впечатление, что большинству из них непросто справляться с согласованием содержимого. Тогда было бы разумнее считать XML форматом API по умолчанию, или, возможно, позволить потребителям указывать, какой формат они хотят использовать при регистрации на портале для разработчиков. Невозможность передать простой заголовок выглядит довольно смешно, но такое может случиться.

Не принимайте как должное, что все потребители могут делать то, к чему вы привыкли. Потребители могут иметь технические ограничения, например программное обеспечение COTS не может добавлять заголовки к HTTP-запросу, но есть много других вариантов. У некоторых потребителей может не быть возможности использовать какой-либо HTTP-метод, кроме `GET` или `POST`. В разделе 10.1 вы видели, что мобильные приложения могут быть ограничены возможностями сети. А в разделе 11.1 мы говорили о веб-хуках. Не все потребители могут с легкостью реализовать это.

Чтобы избежать слишком позднего обнаружения существующих практик или ограничений, которые противоречат тому, к чему вы привыкли, необходимо проявить сочувствие к вашим целевым потребителям. Не стесняйтесь говорить с ними, задавать им вопросы, обсуждать с ними свои проекты – и вы не пожалеете об этом.

Конечно, как обсуждалось в разделе 10.3.8, все это не должно выполняться в ущерб юзабилити и возможности повторного использования. Не пытайтесь угодить нескольким потребителям с очень специфическими потребностями с помощью одного API. Вместо этого рассмотрите возможность создания различных слоев API или предоставьте потребителям возможность создавать собственные BFF API.

### 11.2.2 Тщательно учитываем ограничения поставщика

В разделе 2.4 вы научились разрабатывать API, не обнажая при этом перед потребителями сугубо внутренние проблемы. Но избегать обнажения точки зрения поставщика не означает носить шоры и полностью игнорировать ее. При проектировании API мы должны учитывать то, что происходит позади API, чтобы предложить решение, которое будет не только удобным для использования, но и реализуемым. На рис. 11.9 показано несколько примеров.

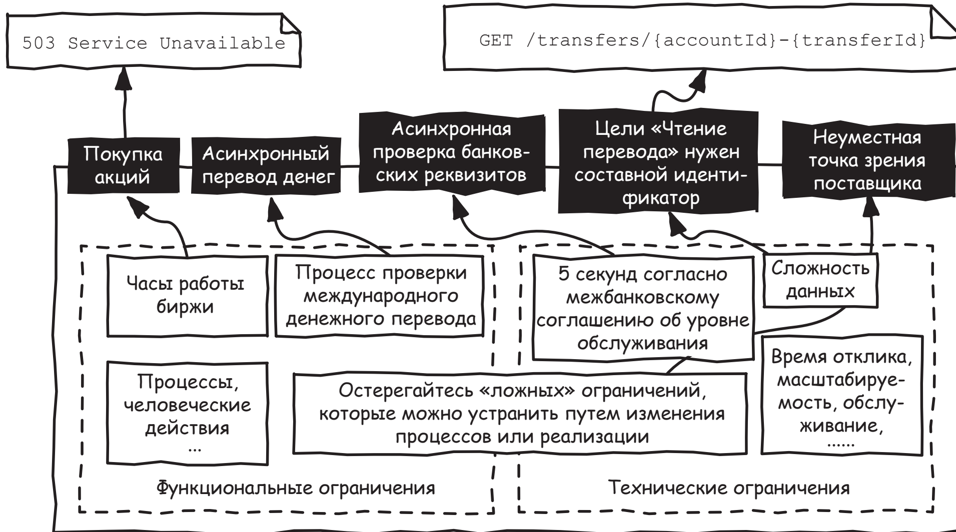



**Рис. 11.9.** Примеры ограничений поставщика

Если банковская компания хочет предоставлять связанные с торговлей цели, такие как покупка акций или продажа акций, ее проектировщики должны знать, что фондовые биржи не всегда открыты, чтобы создать адекватный дизайн. Потребитель, который пытается купить акции на закрытой фондовой бирже, должен получить ошибку, сообщающую ему о том, что операция в данный момент невозможна. Такая ошибка может быть представлена кодом состояния `503 Service Unavailable`. Как вы узнали в разделе 5.2.3, ошибка должна сопровождаться данными, которые помогут потребителю, такими как время открытия фондовой биржи. И, как уже обсуждалось в разделе 5.3.2, также было бы полезно добавить цели для перечисления имеющихся фондовых бирж или предоставления подробной информации о рыночном календаре фондовой биржи и часах торгов, чтобы предотвратить такие ошибки.

**ПРИМЕЧАНИЕ.** Если вы не позаботитесь о функциональных ограничениях, получившийся API будет неполным и менее удобным для потребителя.

Торговля акциями выходит за рамки нашего банковского API, но мы видели пример функционального ограничения, которое повлияло на

проектирование API в разделе 11.1.1. Международный денежный перевод сверх указанной суммы должен быть подтвержден человеком; и, следовательно, он не может быть представлен базовой целью запрос/ответ: вместо этого нужно использовать асинхронное представление. Возможно, стоит выяснить, можно ли найти решение, позволяющее не использовать этап проверки человеком при международных денежных переводах. Это позволило бы нам обеспечить синхронную цель в реальном времени и более удобную для потребителя, вместо асинхронной.

Но ограничения поставщика не только функциональны – они также могут быть техническими. Сервис проверки банковских реквизитов, который вы видели в разделе 11.2.1, опирается не только на саму банковскую компанию, но и на другие банки. Чтобы убедиться, что банковский счет существует в другом банке, банковская компания должна связаться с этим банком. Этот сервис использует асинхронную стандартизированную службу межбанковских сообщений. В соглашении об уровне услуг этой межбанковской системы верификации банковских реквизитов говорится, что проверка должна занимать менее пяти секунд.

Построение синхронной API-цели запрос/ответ поверх такой системы может быть проблематичным. Это может означать, что потребителю может потребоваться ждать ответа пять секунд, что может показаться вечностью (особенно мобильным потребителям). Следовательно, вместо синхронной цели запрос/ответ API должен позволить потребителям отправлять запрос на верификацию, а затем получать результат позже или даже получать уведомление через веб-хук о том, что результат доступен.

Некоторые ограничения могут быть довольно тривиальными, например: «Ой, у нас нет уникального идентификатора для идентификации переводов». Не паникуйте; в этом случае вы можете использовать составные идентификаторы, состоящие из различных идентификаторов или значений, необходимых для идентификации чего-либо. В этом примере в запросе `GET /transfers/{accountId}-{transferId}` можно использовать идентификаторы `accountId` и `TransferId`. Обратите внимание, что, если этот составной идентификатор относится только к вам, ваш видимый контракт интерфейса может быть непрозрачным и отображать только `GET /transfers/{ID}`, причем значение `id` представляет собой составной идентификатор, возвращаемый целью «Перечислить переводы».

Как и функциональные ограничения, технические ограничения на стороне поставщика должны быть подвергнуты сомнению, но подвергнуты *осторожно*. Не обманывайтесь примером технических ограничений, который вы только что видели. В отличие от функциональных ограничений, которые обычно бывают реальными труднорешаемыми проблемами, технические ограничения чаще всего *ложные*. Их можно решить без особых усилий путем внесения изменений в реализацию. Такое небольшое усилие позволяет избежать значительного влияния на проектирование API и, самое главное, на потребителей.

Я не могу сосчитать, сколько раз я слышал такие вещи, как «Мы не можем объединить эти данные; единичные вызовы уже отнимают слиш-

ком много времени!», когда все, что нужно, – это активировать сжатие (см. раздел 10.2.1), добавить отсутствующие индексы в базу данных или оптимизировать запросы к базе данных. Такое простое изменение часто может давать потрясающую производительность, позволяя проектировщикам реализовать предположительно невозможную функцию.

Вот еще один пример. Не так давно в крупных старых компаниях, обнаруживших, что интернет – это не только сайты, использующие методы `POST` и `GET`, вы, возможно, слышали, что использование HTTP-метода `DELETE` невозможно; он блокируется брандмауэрами. Все что было нужно, чтобы решить эту проблему, – поговорить с людьми, отвечающими за безопасность сети, и объяснить новые потребности, чтобы они могли изменить конфигурации брандмауэров, разрешив использовать HTTP-метод, отличный от `POST` и `GET`. (Обратите внимание, что этой специфической проблемы, связанной с HTTP-методом больше нигде не должно быть; по крайней мере, я на это надеюсь!)

Для вас, как для проектировщика API, это означает, что вы должны знать всю цепочку между потребителями и точкой, в которой предоставляется доступ к API, и его фактической реализацией, а также то, что происходит внутри при проектировании, чтобы вы могли как можно скорее заметить технические ограничения и решить эти проблемы либо в рамках реализации, либо путем адаптации дизайна.

Технические ограничения обычно зависят от времени отклика, масштабируемости или доступности базовых систем и сетевых ограничений. Например, довольно неприятно обнаружить во время промышленной эксплуатации, что запрос API занимает более двух секунд, и эту проблему можно было бы решить, добавив больше ЦП и оптимизировав реализацию или в крайнем случае API. Потребители могут быть неприятно удивлены, обнаружив, что ваш API недоступен в течение 15 мин каждый день в полночь благодаря ежедневной перезагрузке или процедуре резервного копирования. Неприятно осознавать, что каждая из ваших тщательно созданных ошибок с кодом `5XX` заменяется универсальной ошибкой `500 Server Error`, телом которой является HTML-страница, благодаря рьяному старомодному брандмауэру или неправильно настроенному API-шлюзу.

Важно помнить, что проектирование API требует глубокого понимания того, что на самом деле происходит до и после выполнения запросов, чтобы вы могли определить возможные функциональные или технические ограничения. Любое потенциальное ограничение должно быть подвергнуто сомнению, потому что возможно полностью или частично решить проблему посредством реализации (в широком смысле), не влияя на дизайн API. Только при тщательном рассмотрении проблемы вы сможете соответствующим образом адаптировать дизайн API, если это будет необходимо.

Функциональные или технические ограничения со стороны поставщика могут принимать различные формы и иметь столько же решений, основанных на адаптированной передаче данных или адекватных целях, свойствах ввода/вывода или обработке ошибок. Но каким бы ни было

решение, вы всегда должны скрывать, насколько это возможно, точку зрения поставщика для предоставления простых для понимания и простых в использовании API.

## 11.3 Выбор стиля API в соответствии с контекстом

Когда вы освоили или привыкли использовать такой инструмент, как молоток, очень заманчиво воспринимать все проблемы, как гвозди. Это когнитивное отклонение, именуемое законом инструмента, законом молотка или законом Маслоу (https://en.wikipedia.org/wiki/Law_of_the_instrument). У такого отклонения также может быть другой эффект: пользователи отвертки могут думать, что отвертка – лучше, чем молоток, в то время как пользователи молотка могут думать обратное. Это можно назвать *народным законом фанатов*.

Но молоток не решит всех проблем, а отвертка не лучше молотка; каждый инструмент так же полезен, как и другой, но в разных контекстах. Эта книга о проектировании веб-API, а не о столярном деле или деревообработке, но те же проблемы применимы и к технологической индустрии. Выбор того, какой инструмент (инструменты) вы будете использовать для проектировании удаленного API, нельзя делать исходя из того, к чему вы привыкли, что модно или основываясь на своих личных предпочтениях; это нужно делать в соответствии с контекстом. А чтобы выбрать правильный инструмент, вам нужно знать больше, чем об одном.

Веб-API легко можно превратить в API + единичный и синхронный запрос/ответ + REST + HTTP 1.1 + JSON, что в настоящее время является одним из наиболее часто используемых способов активации обмена данными по типу «software-to-software» для предоставления доступа к целям, отвечающим нуждам целевых пользователей. Поэтому разработчики API могут испытывать желание использовать этот набор инструментов во всех ситуациях, во всех контекстах. В этой книге этот набор инструментов используется только для раскрытия фундаментальных принципов проектирования API, которые можно использовать при проектировании других типов удаленных API.

Мы уже познакомились с некоторыми другими инструментами, которые можно добавить в наши наборы инструментов, чтобы использовать их в соответствующих контекстах. Например, в разделе 6.2.1 вы видели, что JSON был не единственным возможным форматом данных для API; Вы можете использовать XML, CSV, PDF и многие другие форматы. В разделе 11.2.1 вы также видели, что иногда использование формата JSON даже может привести к обратным результатам в контексте, если потребители привыкли к существующему стандартизированному формату XML. В разделе 10.3.6 вы узнали, что REST API не единственный вариант при создании веб-API. Использование языка запросов может дать больше гибкости при запросе данных (но меньше возможностей для кеширования). В разделе 11.1 вы обнаружили, что синхронный механизм запрос/ответ от потребителя к поставщику не является единственным способом обеспечения передачи данных между двумя системами. Мы

можем создавать асинхронные цели, уведомлять потребителей о событиях, осуществлять потоковую передачу данных и даже обрабатывать несколько элементов за один вызов. А в разделе 10.2.1 вы узнали, что вместо старого доброго протокола HTTP 1.1 можно использовать HTTP 2.

Мы уже знаем, что контекст играет важную роль в выборе инструментов, и уже знаем о нескольких инструментах. Но, будучи проектировщиками API, как и проектировщиками программного обеспечения и систем в целом, нам необходимо расширить свою точку зрения, чтобы быть уверенными, что мы избегаем закона инструмента. Для этого мы рассмотрим некоторые альтернативы REST API и веб-API в этом разделе.

### 11.3.1 Сравнение API на базе ресурсов, данных и функций

На момент написания этой книги существует три основных способа создания веб-API: REST, gRPC и GraphQL. Будут ли они использоваться через пять или десять лет? Будут ли они прежними? Время покажет.

Один из них лучше других? Нет! Все зависит от потребностей и контекста. Подходы, показанные на рис. 11.10, представляют три различных видения API: REST ориентирован на ресурсы, gRPC – на функции, а GraphQL – на данные, и у каждого из них есть свои плюсы и минусы.

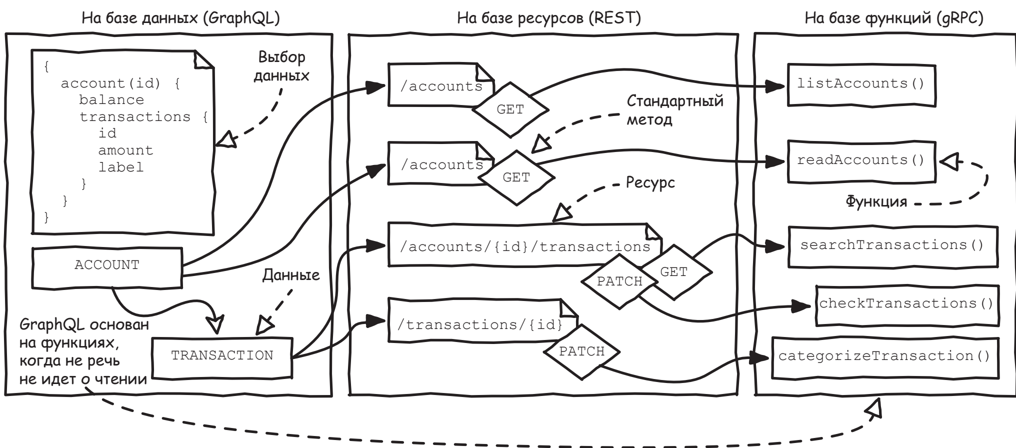



Рис. 11.10. Сравнение API на базе ресурсов, данных и функций

Теперь вы должны знать, что такое REST API. Как вы видели в этой книге и особенно в разделе 3.5.1, REST API – или RESTful API – это API, который соответствует (или, по крайней мере, пытается соответствовать) архитектурному стилю REST, введенному Роем Филдингом[1]. Такой API основан на ресурсах и использует преимущества базового протокола (в данном случае протокола HTTP). Его цели представлены использованием стандартных методов HTTP с ресурсами, а результаты – стандартными кодами состояния HTTP.

[1] См. его кандидатскую диссертацию «Архитектурные стили и проектирование сетевых программных архитектур» по адресу https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

В банковском API чтение данных счета может быть представлено запросом `GET /owners/123`, возвращающим код состояния `200 OK` вместе со всеми данными клиента, если этот владелец 123 существует, и `404 Not Found` – если нет. Обновление VIP-статуса одного и того же владельца можно выполнить с помощью запроса `PATCH /owners/123`, тело которого будет содержать новое значение.

Использование существующего протокола способствует согласованности и делает API предсказуемыми, как вы видели в разделе 6.1. Увидев любой ресурс, потребитель может попытаться использовать HTTP-метод `OPTIONS`, чтобы определить, что с ним можно сделать, или даже попробовать метод `GET`, чтобы прочитать его или методы `PUT` или `PATCH`, чтобы обновить его. Даже самый неясный код состояния `4XX` будет восприниматься всеми потребителями как ошибка на стороне потребителя. Такой API также может использовать все существующие функции HTTP, такие как кеширование и условные запросы; не нужно изобретать велосипед. Также можно добавить возможности потоковой передачи от сервера к потребителю с помощью технологии SSE (см. раздел 11.1.3). Но это упростит проектирование API.

В этой книге вы видели, что, даже если протокол HTTP предоставляет какой-то фреймворк, он не может волшебным образом помешать нам создавать ужасные REST API. Проектировщики по-прежнему должны выбирать пути к ресурсам (`/owner` или `/owners`?) и решать, как представлять данные, выдавать информативные сообщения об ошибках или успехах за пределами HTTP-состояний и др.

Фреймворк gRPC был создан компанией Google. g значит Google, а *RPC* – это Remote Procedure Call, Удаленный вызов процедур. RPC API просто предоставляет доступ к функциям.

В API на базе функций чтение владельца 123 можно выполнить, вызвав функцию `readOwner(123)`, а обновление VIP-статуса этого владельца можно совершить путем вызова функции `updateOwner(123, {«vip»: true})`. Фреймворк gRPC использует протокол HTTP 1.1 или 2 в качестве транспортного уровня, не используя его семантику. Он не предоставляет никакого стандартного механизма кеширования. Обратите внимание, что он может использовать протокол HTTP 2 для двунаправленной и потоковой передачи данных, а также формат данных Protocol Buffer, который является менее многословным по сравнению с XML или JSON (вы также можете использовать этот формат в REST API).

В то время как в случае API на базе ресурсов базовый протокол обеспечивает какой-то фреймворк, в особенности для описания того, какие действия предпринимаются и каков результат, в API на базе функций, как правило, проектировщики сами выбирают собственную семантику почти для всего. Итак, как бы вы представили такую цель, как «Перечислить владельцев»? Должна ли это быть функция `listOwners()`, `readOwners()` или `retrieveOwners()`? То же самое касается и изменения данных. Должен ли API предоставлять функцию `saveOwner()` или `updateOwner()`?

Что касается ошибок, фреймворк gRPC предоставляет стандартную модель ошибок, включая несколько стандартных кодов, которые сопо-

ставимы с кодами состояния HTTP (https://cloud.google.com/apis/design/errors). Например, при вызове функции `readOwner(123)` может быть возвращен код `NOT_FOUND` (аналогичный `404 Not Found`) вместе с сообщением `Owner 123 does not exist`. Модель ошибок можно дополнить добавочными данными, чтобы обеспечить более информативное ответное сообщение. Как и в случае с REST API, проектировщики сами выбирают, как это сделать (см. раздел 5.2.3), а также как представлять данные.

Мы кратко рассмотрели GraphQL в разделе 10.3.6; это язык запросов для API, созданных Facebook. GraphQL API в основном обеспечивает доступ к схеме данных, позволяя потребителям получать именно те данные, которые им нужны. Он не зависит от протокола, т. е. можно использовать любой протокол, который позволяет нам отправлять запросы и получать ответы; но, поскольку протокол HTTP является наиболее распространенным, обычно выбирают его.

Как и gRPC, GraphQL не предоставляет никакого стандартного механизма кеширования. Запрос `POST /graphql` с запросом `{ "query": "{ owner(id:123) { vip } }" }` в его теле будет возвращать только VIP-статус владельца 123. А когда дело доходит до создания или обновления данных, GraphQL ведет себя, как и любой RPC API. Он использует функции, которые называются *мутациями*. Обновление VIP-статуса владельца 123 потребовало бы от нас вызова мутации `updateOwner`, которая принимает идентификатор владельца и объект `owner`, содержащий новый VIP-статус.

GraphQL также поставляется со стандартной моделью ошибок, которую можно расширить. В листингах 11.9 и 11.10 показаны запрос и ответ со стандартной ошибкой соответственно.

**Листинг 11.9. Запрос в GraphQL**

```
{
   owner(id: 123) {
      vip
      accounts {
         id
         balance
         name
      }
   }
}
```

**Листинг 11.10. Ответ с ошибкой**

```
{
   "errors": [
      {
         "message": "No balance available for account with ID 1002.",
         "locations": [ { "line": 6, "column": 7 } ], ①
         "path": [ "owner", "accounts", 1, "balance" ] ②
      }
```

```
  ],
  "data": {
    "owner": {
      "vip": true,
      "accounts": [
        {
          id": "1000",
          "balance": 123.4
          "name": "James account"
        },
        {
          "id": "1002",
          "balance": null, ③
          "name": "Enterprise account"
        }
      ]
    }
  }
}
```

① **Указывает на ошибку в запросе.**
② **Указывает на свойство результата, на которое повлияла ошибка.**
③ **Фактическое свойство, затронутое ошибкой.**

Запрос, показанный в листинге 11.9, запрашивает VIP-статус владельца 123 и идентификаторы счетов, балансы и имена. К сожалению, как показано в листинге 11.10, получить баланс по второму счету нельзя. Стандартная модель ошибок содержит для каждой ошибки читабельное сообщение (`message`), возможные источники ошибки в запросе GraphQL находятся в свойстве `locations` (`balance` находится на шестой строке и начинается с седьмого символа запроса) и необязательное свойство `path` к затронутому свойству – в возвращаемых данных (нулевой баланс находится в `data.owner.accounts[1] .balance`).

Такая ошибка, по-видимому, является ошибкой поставщика, а не потребителя, но это не указано. Проектировщики сами выбирают, как добавить информацию в эту стандартную модель ошибок, чтобы обеспечить полностью информативное ответное сообщение. И очевидно, как и в REST API и gRPC, они сами выбирают, как проектировать модель данных.

С точки зрения проектирования, видно, что эти три различных способа создания API имеют три разных способа представления целей: ресурсы (REST), функции (gRPC, а также создания и модификации в GraphQL) и данные (чтения в GraphQL). По сути, представление любой цели чтения возможно в любом из этих стилей API. Когда речь идет о создании, изменении, удалении или выполнении целей, они могут быть представлены с помощью пары ресурс/метод или функцией. Каждый подход включает более или менее стандартизированные элементы, способствующие согласованности и, следовательно, облегчающие удобство использования и проектирование.

**ПРИМЕЧАНИЕ.** API, строго следующий правилам базового протокола, является наиболее последовательным API из коробки.

Но независимо от предоставленного фреймворка проектировщикам по-прежнему нужно проделать большую работу для проектирования достойных API. Независимо от стиля API, который они выбирают, им все равно нужно определить пользователей, цели, входные и выходные данные и ошибки и выбрать наиболее подходящие из возможных ориентированные на потребителя представления, избегая точки зрения поставщика.

С технической точки зрения у нас есть три разных API-инструмента или технологии, которые можно использовать по протоколу HTTP. Использование протокола HTTP важно, потому что он широко распространен, и обычно для размещения или использования API на базе HTTP не требуется много, если они есть, изменений в вашей инфраструктуре. Однако между этими тремя инструментами существуют некоторые отличия.

REST API используют протокол HTTP и могут использовать такие функции, как согласование содержимого, кеширование и условные запросы. GraphQL и gRPC не предоставляют таких механизмов, но у них есть некоторые другие интересные возможности. Благодаря использованию HTTP 2 и формата данных ProtoBuf, API на базе gRPC могут обеспечить высокую производительность. Они также обеспечивают потоковую и двунаправленную передачу данных между потребителем и поставщиком. (Обратите внимание, что REST API могут обеспечивать одностороннюю потоковую передачу данных от поставщика к потребителю с помощью технологии SSE.) И, как видно из раздела 10.3.6, возможности запросов GraphQL позволяют потребителям получать все данные, которые им нужны (и только те данные, которые им нужны) в одном запросе, но в ущерб возможностям кеширования.

Что касается контекста поставщика и особенно реализации, очевидно, у вас нет большого контроля над запросами, которые могут выполняться потребителями в API на базе данных. В системах с неограниченной масштабируемостью слишком большое количество сложных запросов может привести к нагрузке, превышающей ту, что поддерживают базовые системы, и ужасно долгому времени отклика, если реализация не готова предотвратить это. В случае с API на базе ресурсов или функций таких проблем довольно легко избежать. Поскольку поведение каждой цели обычно предсказуемо, запрашиваемые системы известны, а ограничение скорости можно использовать для защиты базовых систем. Вы можете указать, что каждый потребитель может делать не больше x запросов в секунду через API, и даже специализировать это ограничение скорости в зависимости от потребителя и/или цели.

В случае с API на базе данных можно ограничить количество запросов или их размер, но это будет бессмысленно, поскольку это не помешает выполнению неожиданно сложных запросов. Вы можете ограничить количество узлов в запросе (содержащем один или несколько запросов) или принимать только предварительно зарегистрированные запросы, но это будет сделано в ущерб гибкости, что сделает выбор API на базе данных практически бесполезным. Во всех случаях (REST, gRPC,

GraphQL) рекомендуется ограничить количество элементов, возвращаемых по умолчанию в списках.

Итак, какой же подход использовать? Такой выбор нельзя сделать до анализа вашего контекста и потребностей. Как только вы узнаете, кто ваши потребители, и поймете их контекст, цели, которые им нужны, и то, как они будут использоваться, и поймете контекст поставщика, вы можете решить, какой API будет наиболее подходящим. Хотя все контексты будут отличаться, в настоящее время практическим правилом является выбор REST по умолчанию.

При наличии очень специфических потребностей, которые нельзя удовлетворить хорошо спроектированным REST API, можно попробовать GraphQL или gRPC.

Выбор REST по умолчанию можно рассматривать как пример закона инструмента или народного закона фанатов, но подход с использованием REST способен удовлетворить большинство потребностей. Это наиболее распространенный способ создания API, и большинство проектировщиков к нему привыкли (помните раздел 11.2.1?). Выбирайте GraphQL для закрытых API в мобильных средах, только если хорошо спроектированный REST API, размещенный в хорошо сконфигурированной среде, невозможно использовать (см. раздел 10.2) и если

- вам действительно нужны расширенные возможности выполнения запросов;
- вы не планируете публиковать свой API или использовать его совместно с партнерами;
- вас не интересует кеширование;
- вы уверены, что сможете защитить базовые системы посредством реализации или бесконечной масштабируемости.

И наконец, выбирайте gRPC API для передачи данных «внутреннее приложение-внутреннему приложению», только если миллисекунды действительно имеют значение, если вас не интересует кеширование или вы хотите обрабатывать его, не полагаясь на HTTP, и если вы не планируете делать API закрытым или использовать его совместно с партнерами. Также имейте в виду, что этот выбор не может быть эксклюзивным. В разделе 10.3.8 вы уже видели, что разные слои API могут удовлетворять различным потребностям. Создание мобильного BFF с предоставлением доступа к GraphQL API или более специализированному REST API вполне допустимо. Приложение также может предоставлять доступ к интерфейсу gRPC для внутренних потребителей и к интерфейсу REST для внешних.

### 11.3.2 За границами API на базе HTTP

Будучи проектировщиками API, мы должны знать, что API на базе HTTP (запрос/ответ) не единственный способ обеспечения передачи данных между приложениями. Мы говорили о событиях и потоковой передаче данных в разделе 11.1, но в основном с точки зрения HTTP. При создании системы на основе событий поставщик может уведомлять потребителей

о событиях с помощью веб-хука или протокола WebSub. Оба они используют протокол HTTP. Но это также можно сделать с помощью системы обмена сообщениями, такой как RabbitMQ. Если это для внутренних целей, возможно, более эффективно будет напрямую подключать поставщиков и потребителей к таким инструментам.

При работе с интернетом вещей (IoT) эффективность энергопотребления – ключевая проблема, а двусторонняя передача данных по ненадежным сетям или со спящими устройствами является почти стандартом. Протокол MQTP – это протокол на основе сообщений, разработанный для устранения таких ограничений. При потоковой передаче событий вы можете использовать SSE по протоколу HTTP для передачи данных между поставщиком и потребителем. Но это также можно делать и с использованием протокола WebSocket, который не основан на HTTP (как показано в разделе 11.1.3). И если нужно обрабатывать массивный поток событий, вам может подойти Kafka Streams.

Чтобы рассказать о проектировании и архитектуре систем, основанных на событиях, понадобится целая книга, и мы не преследуем здесь эту цель. Но вы можете по крайней мере воспользоваться тем, что узнали из этой книги, для проектирования уведомлений о событиях и потоков. Следует помнить, что обмен данными по протоколу HTTP не единственный вариант; и в определенных контекстах этого следует избегать любой ценой. В следующей главе вы узнаете о различных типах документации API и о том, как проектировщики могут участвовать в ее создании.

## *Резюме*

- Единичный запрос/ответ, передача данных между потребителем и поставщиком не единственная возможность; вы также можете создавать асинхронные цели, уведомлять потребителей о событиях, осуществлять потоковую передачу данных и обрабатывать несколько элементов за один вызов.
- Проектирование API требует, чтобы мы знали контексты потребителей, включая их сетевое окружение, привычки и ограничения.
- Проектирование API требует, чтобы мы тщательно рассматривали ограничения поставщика, выявляли их на раннем этапе и решали проблемы, не влияя на дизайн (если это возможно) и не адаптируя его (если это невозможно).
- Проектирование API требует от нас игнорировать моду и личные предпочтения. Тот факт, что вам что-то нравится или вы знакомы с определенным инструментом/дизайном/практикой, не означает, что это будет идеальным решением для всех вопросов, касающихся проектирования API.

# 12 Документирование API

**В этой главе мы рассмотрим:**

- справочную документацию;
- руководства пользователя;
- спецификации разработчика;
- журналы изменений.

В предыдущих главах вы узнали, что проектирование API – это не просто проектирование удобных в использовании API, делающих свою работу. Мы должны заботиться обо всем контексте, окружающем API, при его разработке. Но этот контекст выходит за рамки самого API – его интерфейсного контракта, его реализации, а также того, как и кем он используется. Проектировщики API должны участвовать в различных аспектах проектов API, и очень важным является документация.

Лучшие проекты даже самых простых вещей нуждаются в документации. Как показано на рис. 12.1, повседневные предметы могут поставляться с различными типами документации, чтобы помочь пользователям понять, как их использовать, а также помочь людям, отвечающим за создание этих предметов, создать их.

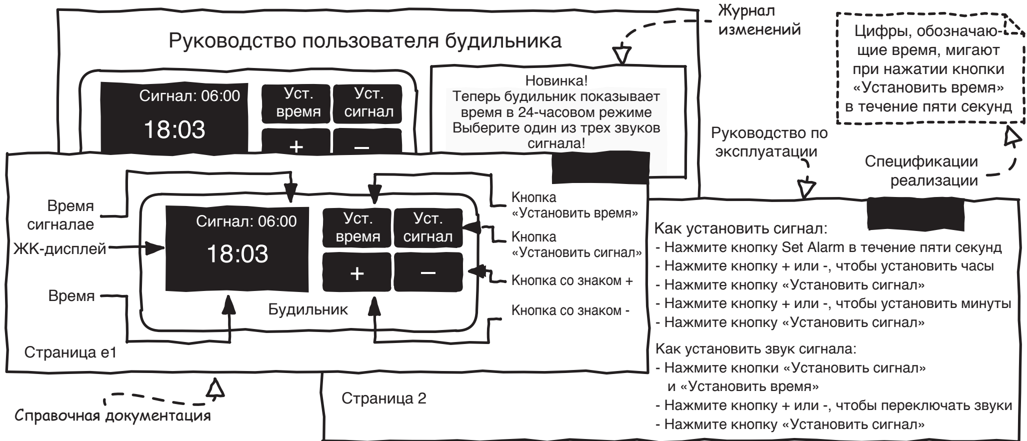



**Рис. 12.1. Различные типы документации**

На стр. 1 руководства пользователя будильника показан рисунок будильника с аннотациями. Благодаря этому пользователи знают, что представляют собой компоненты пользовательского интерфейса и их роли, даже если удобный дизайн устройства делает это довольно очевидным. Но одной этой первой страницы документации недостаточно для работы будильника. Вот почему на стр. 2 показаны различные функции, такие как установка будильника с помощью кнопок **Установить будильник** и **плюс** и **минус**.

Обложка руководства пользователя также является своего рода документацией. В ней говорится «Сейчас время отображается в 24-часовом режиме», чтобы указать на новую функцию, которой не было в предыдущей версии. (Новым пользователям это может быть неинтересно, но станет интересно тем, у кого была более ранняя версия.) Без этой документации большинство пользователей, особенно если у них уже был такой, могло бы использовать будильник благодаря его дизайну. Но некоторые пользователи, абсолютные новички, поначалу могут с трудом догадаться, как им управлять.

Мы упомянули три разных типа документации, которые ориентированы на пользователя, но есть и другой вид документации. Хотя в этом руководстве пользователя содержится вся информация, необходимая для использования будильника, этого недостаточно, чтобы создать его. Например, в руководстве пользователя не указывается, что время на ЖК-экране мигает при нажатии кнопки **Установить время** в течение пяти секунд, но на самом деле это происходит. Люди, отвечающие за создание будильника, получили документацию от его дизайнеров в виде *спецификаций реализации*, описывающих такое поведение. Без такой документации маловероятно, что дизайн будет реализован так, как этого ожидают дизайнеры. Без соответствующей документации все усилия, вложенные в дизайн, могут быть бесполезными.

Будучи проектировщиками API, вам придется создавать или, по крайней мере, участвовать в создании такой документации для API, кото-

рый вы проектируете. Самая известная документация по API – это справочная документация, которая описывает интерфейсный контракт API. В ней перечислены доступные цели и описаны их входные и выходные данные. Это то, что вы должны описать при проектировании API. Этого может быть достаточно для очень простых API, если все варианты использования, выполняемые API, можно выполнить с помощью одной цели. Но если это не так, то только предоставление справочной документации похоже на предоставление списка ингредиентов для рецепта без указания того, что делать с этими ингредиентами, – получить съедобный результат таким образом будет довольно непросто. Вот почему API также должен сопровождаться руководством по эксплуатации, где описываются различные варианты использования и способы их достижения.

Кроме того, как и в любом программном обеспечении, при изменении API, даже если не вносятся критические изменения, целесообразно предоставить журнал изменений с указанием функций, которые были изменены или добавлены. Будучи разработчиком, который знает, какие изменения были внесены, вы должны перечислить их. И последнее, но не менее важное: предоставления описания API может быть недостаточно, для того чтобы кто-то мог его реализовать. Вам также может потребоваться предоставить дополнительные спецификации лицам, отвечающим за реализацию API, чтобы обеспечить ожидаемый результат.

Ваше участие в каждом из этих видов документирования зависит от типа документации, размера вашей компании или команды и типа API (закрытый, партнерский или открытый). В крупной компании и/или команде могут быть технические писатели для создания высококачественной документации; вам нужно лишь обеспечить поддержку и исходные данные.

Прибегать к услугам технических писателей особенно важно, когда речь идет о документации для открытых API, ориентированной на потребителя; для написания удобной в использовании и подходящей для клиента документации требуются специалисты, чтобы обеспечить высококачественный опыт разработчика. В случае с закрытыми API ожидания могут быть ниже и документация может быть менее привлекательной, но опыт разработчика по-прежнему должен вызывать серьезную озабоченность. Документация для внутренних разработчиков должна быть как минимум читаемой и исчерпывающей. У исчерпывающего документирования API есть хороший побочный эффект – так мы тестируем дизайн. Если вы не можете задокументировать, как использовать API или как его реализовать, это может быть признаком плохого дизайна.

В этой главе мы узнаем, что может содержаться в справочной документации, руководствах по эксплуатации, спецификациях реализации и журналах изменений, и мы, как проектировщики API, можем внести в них свой вклад, воспользовавшись своей работой при проектировании API. То, что вы научитесь делать здесь, может быть достаточно для полного документирования закрытых API; в случае с партнерскими API, ориентированными на потребителя, или открытых API ваша работа станет хорошим подспорьем для более опытных технических писателей.

## 12.1 Создание справочной документации

Справочная документация API, подобная той, что показана на рис. 12.2, похожа на схему будильника с аннотациями на рис. 12.1: в ней перечисляется и описывается каждый доступный компонент интерфейса.

**Рис. 12.2. Справочная документация, сгенерированная из файла спецификации OpenAPI с использованием инструмента с открытым исходным кодом ReDoc**

Для API компоненты как минимум являются доступными целями, а также их входными и выходными данными как для случаев успеха, так и ошибок. (В случае с будильником этими компонентами были его кнопки и ЖК-экран.) Документация по API также должна содержать простое описание и предоставлять информацию о безопасности. Вся эта информация может быть записана в любом формате, от простого текстового файла до страницы на «Википедии». Некоторые даже осмеливаются использовать электронные таблицы (пожалуйста, не делайте этого!). С этой задачей может справиться любой пользовательский формат, но вы видели в разделе 6.1.3, что при проектировании API лучше использовать стандарты, и это также относится и к созданию документации API.

О некоторых из этих стандартов вы узнали в главе 4. Форматы описания API, такие как спецификация OpenAPI, являются идеальным компаньоном, когда вы хотите создать справочную документацию по API. Эти форматы созданы для описания того, что необходимо в такой документации; и, как видно из раздела 4.1.2, их можно легко хранить, управлять версиями и, что наиболее важно, использовать для создания удобного для человека представления. Инструмент, используемый для генерации справочной документации на основе файла спецификации OpenAPI, показанного на рис. 12.2, называется ReDoc.

### Запуск ReDoc CLI

Все скриншоты документации API из этой главы были созданы с помощью утилиты командной строки `redoc-cli` (https://github.com/Rebilly/ReDoc/tree/master/cli) с использованием приведенной ниже командной строки:

```
redoc-cli serve <path to OpenAPI file> --options.
➥ showExtensions
```

Эта утилита также позволяет создавать автономную документацию HTML с помощью этой строки:

```
redoc-cli bundle <path to OpenAPI file> --options.
➥ showExtensions
```

Это лишь некоторые из возможностей ReDoc. Документация к ней (https://github.com/Rebilly/ReDoc) предоставляет всю необходимую информацию для ее использования и интеграции в любые существующие веб-приложения.

ReDoc – это один из множества инструментов. Многие инструменты API, особенно порталы разработчиков API, нативно понимают спецификацию OpenAPI (и другие спецификации) и поэтому могут генерировать такие визуализации без необходимости писать какой-либо код. Но одного формата недостаточно. Если описание вашего API не содержит всей необходимой информации, сгенерированная документация будет неполной, даже если эти инструменты сами смогут угадать что-то.

В последующих разделах мы возьмем в качестве иллюстрации спецификацию OpenAPI и ReDoc, но вы можете использовать другие форматы описания API и инструменты рендеринга. Здесь важно помнить, какая информация необходима в справочной документации и чего вам следует ожидать от описания форматов API и средств визуализации. Мы начнем с документирования модели данных, а затем целей (пути и HTTP-методы, ввод, вывод). После этого займемся безопасностью и наконец увидим, как добавить полезную информацию о самом API.

## 12.1.1 Документирование моделей данных

На рис. 12.3 и 12.4 показана подробная модель данных и пример параметра тела запроса, соответственно, необходимого для создания денежного перевода.

REQUEST BODY SCHEMA: application/json

| | |
|---|---|
| source<br>required | string 15 characters /^\d{15}$/<br>Source account number |
| destination<br>required | string 15 characters /^\d{15}$/<br>Destination account number |
| amount<br>required | number > 0 |
| date | string <date><br>Execution date for a delayed transfer or first execution date for a recurring one |
| occurrences | integer [ 2 .. 100 ]<br>Number of times a recurring transfer will be executed |
| frequency | string<br>Enum: "WEEKLY" "MONTHLY" "QUARTERLY" "YEARLY"<br>Frequency of recurring transfer's execution |

Рис. 12.3. Справочная документация по модели данных

Request samples

Payload

application/json

Copy Expand all Collapse all

```
{
    "source": "000534115776675",
    "destination": "000567689879878",
    "amount": 456.2,
    "date": "2019-05-01",
    "occurrences": 2,
    "frequency": "WEEKLY"
}
```

Рис. 12.4. Пример справки

Справочная документация на рис. 12.3 показывает, что эта модель данных состоит из свойств source, destination, amount, date, occurrences и frequency. Все они имеют тип string, за исключением свойства amount, которое является числом, и occurrences: это целое число; кроме того, source, destination и amount являются обязательными свойствами. Это самая основная информация, необходимая в справочной документации, как и при описании API на этапе проектирования (см. раздел 3.3.1).

Но эта документация не только предоставляет минимум сведений: она также дает полезные функциональные и технические описания и примеры. В ней содержится подробная информация о формате и значении каждого свойства. Свойства `source` и `destination` должны быть длиной 15 символов и содержать только цифры, согласно регулярному выражению `/^\d{15}$`. amount (исключительно положительное число), а значение свойства occurrences должно быть от 2 до 100. Возможные значения свойства `frequency` – `"WEEKLY"`, `"MONTHLY"`, `"QUARTERLY"` и `"YEARLY"`. Пример, показанный на рис. 12.4, помогает нам визуализировать, как выглядит каждое свойство.

Кроме того, описания `date`, `occurrences` и `frequency` дают полезную информацию о том, как можно использовать эти свойства для создания отложенных или регулярных переводов. Обратите внимание на то, что у свойства `amount` нет описания, потому что его имя и контекст проясняют, что оно определяет сумму, которую нужно перевести с исходного счета на счет назначения.

Вся эта информация поступает из JSON-схемы, определенной в базовом файле спецификации OpenAPI (две выдержки показаны в листингах 12.1 и 12.2). Вы уже узнали в разделе 4.3.2, как определить имя, тип и описание свойства, как указать, если свойство является обязательным и как привести пример.

**Листинг 12.1. Очень полное описание свойства с примером**

```
components:
  schemas:
    TransferRequest:
      description: A money transfer request
      required:
        - source
        - destination
        - amount
      properties:
        source:
          type: string
          description: Source account number
          minLength: 15 ①
          maxLength: 15 ②
          pattern: ^\d{15}$ ③
          example: "000534115776675" ④
```

① Минимальная длина свойства source.
② Максимальная длина свойства source (регулярное выражение).
③ Формат свойства source.
④ Пример значения.

Длина свойства `source` равна 15, поскольку оба значения `minLength` и `maxLength` равны 15. Его точный формат (строка, состоящая из 15 цифр)

определяется шаблоном (pattern), содержащим регулярное выражение ^\d{15}$ (например, 000534115776675).

ReDoc может угадать длину свойства на основе регулярного выражения, но не все инструменты используют эту информацию. Значение свойства source, показанное на рис. 12.4, взято из примера, приведенного в листинге 12.1, но инструменты документирования также могут угадывать значения примеров на основе описаний, как показано в этом листинге.

**Листинг 12.2. Инструменты документирования используют описания для генерации примеров**

```
[...]
date:
  type: string
  format: date ①
  description: | Execution date for a delayed transfer
  ➥ or first execution date for a recurring one
[...]
frequency:
  type: string
  description: Frequency of recurring transfer's execution
  enum: ②
    - WEEKLY
    - MONTHLY
    - QUARTERLY
    - YEARLY
```

① **Свойство date представляет собой строку с использованием формата даты (ГГГГ-ММ-ДД); используется в документации сегодняшняя дата.**

② **Возможные значения свойства frequency; в документации показано случайное значение.**

Свойство date использует формат date, а это означает, что значение будет иметь формат YYYY-MM-DD: например, 2019-03-23. Возможные значения свойства frequency ("WEEKLY", "MONTHLY", "QUARTERLY" и "YEARLY") определены в перечислении. Для свойств date и frequency примеров нет, но в справочной документации есть несколько примеров: они были сгенерированы на основе JSON-схемы.

Для свойства date, являющегося датой, ReDoc просто использует сегодняшнюю дату (2019-05-01 на рис. 12.4), а для свойства frequency – случайное значение из перечисления (WEEKLY на рис. 12,4). Выясните самостоятельно, как были описаны другие свойства.

### Подробнее о спецификации OpenAPI

Как было упомянуто в разделе 4.1.1, можно легко изучить весь формат OpenAPI, используя мою карту OpenAPI и читая ее спецификацию[1]. Обратите внимание, что существуют специальные дополнительные свойства JSON Schema, которые можно использовать для точного описания любых атомарных типов, и также массивы (например, сколько элементов можно использовать в массиве).

[1] См. https://openapi-map.apihandyman.io/ и https://github.com/OAI/OpenAPI-Specification/tree/master/versions.

Как видите, базовую, но ценную справочную документацию для модели данных API можно создать, просто повторно использовав работу, выполненную во время проектирования API. Простого перечисления свойств, их типов и, если они являются обязательными, даже без предоставления примеров или подробных описаний, может быть достаточно для простых моделей данных. Потребители могут не поблагодарить вас за предоставление такой базовой справочной документации, но будьте уверены, что они проклянут вас, если вы вообще не предоставите ее.

Если вы добавите примеры и, что более важно, подробные и релевантные описания, включая как машиночитаемые (например, форматы или возможные значения), так и удобные для восприятия человеком описания, итоговая справочная документация будет полезна всем, кто использует ее как на стороне потребителя, так и на стороне поставщика. Не нужно использовать описания типа «Капитан очевидность», например «`amount`: сумма перевода» или, что еще хуже, «`amount`: сумма».

**ПРИМЕЧАНИЕ.** Соответствующее читабельное описание объясняет характер свойства, в том числе его роли и связь с другими свойствами и когда оно используется.

Если указать имя, тип и контекст свойства, достаточные, для того чтобы пользователь понял его значение, добавлять описание нет необходимости. Если формат описания API и инструмент визуализации, которые вы используете, поддерживают отформатированные удобочитаемые описания (Markdown, например), не стесняйтесь использовать эту функцию, чтобы длинные описания было легче читать.

### *12.1.2 Документирование целей*

Справочная документация цели описывает назначение цели, что необходимо для ее использования, какие ответные сообщения получают потребители в случае успеха или неудачи и входит ли она в группу целей. Например, на рис. 12.5 представлен обзор цели «Перевести деньги».

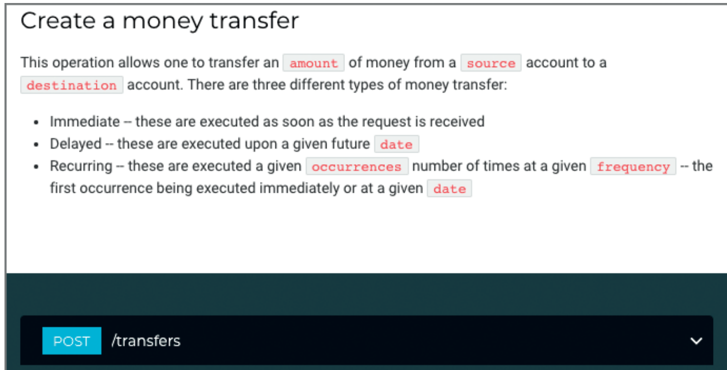



**Рис. 12.5.** Обзор цели «Перевести деньги»

Как видите, цель представлена запросом `POST /transfers`. Согласно описанию, она позволяет пользователям создавать мгновенные, отложенные или регулярные денежные переводы. Каждый тип перевода объясняется с функциональной точки зрения, и также дается информация об использовании различных свойств, которые могут быть общими для всех типов или специфическими для одного из них. Для всех типов переводов требуется сумма, исходный и целевой счет. Также видно, что отложенный перевод выполняется в данную будущую дату, а для регулярного перевода требуется дата, а также число вхождений и частота.

Всю информацию, представленную в этом описании, можно получить путем анализа модели данных тела запроса, ранее показанной на рис. 12.3, но лучше предоставить более удобочитаемое описание, вместо того чтобы просто сказать «Создать денежный перевод» или «Создать немедленный, отложенный или регулярный денежный перевод». Такое описание делает документацию очень удобной для пользователя, но можно сделать еще лучше. Посмотрите на образцы запросов, показанные на рис. 12.6.

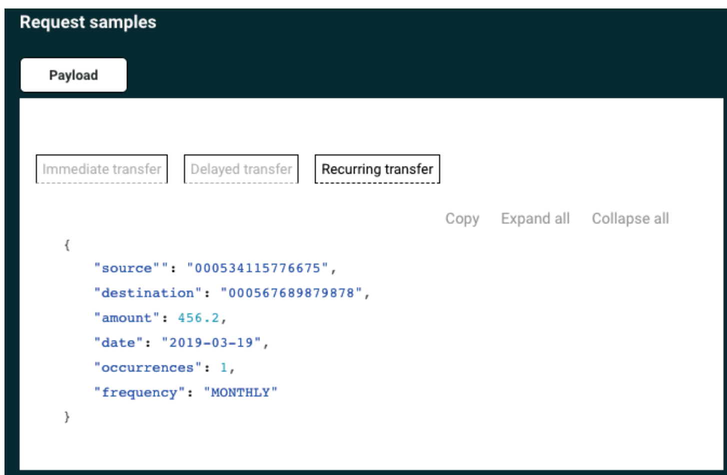



**Рис. 12.6.** Образцы запросов

Здесь есть вкладки **Мгновенный перевод**, **Отложенный перевод** и **Регулярный перевод**, причем выбрана последняя. В этой справочной документации приводится пример запроса для каждого типа денежного перевода. Это необходимая вещь! Пользователям не нужно думать, чтобы получить практически готовые примеры для различных вариантов использования; они просто должны настроить предоставленные значения. Это касается входных данных, но справочная документация API также должна содержать подробную информацию о возможных ответах, как показано на рис. 12.7.

Для каждого ответа имеется удобочитаемое описание и описание тела ответа (в том же формате, что и тело запроса). Справочная документация показывает, что в случае успеха может быть возвращен код состояния `201` или `202`. Ответ `201 Created` возвращается, когда перевод принят, а свойство `date` предоставлено не было (для мгновенных или регулярных переводов с первым вхождением, выполненным немедленно).

Responses

201 Immediate or recurring transfer executed

202 Delayed or recurring delayed transfer accepted

400

The transfer is rejected due to an error in the request properties or an insufficient balance. Each error provides the property `source` of the error along with a human-readable `message` and its `type`:

- MANDATORY_PROPERTY: The property indicated in `source` is missing
- INVALID_FORMAT: The format of the property indicated in `source` is invalid
- INVALID_VALUE: The value of the property indicated in `source` is invalid
- INSUFFICIENT_BALANCE: The `amount` property is higher than the `source` account balance

RESPONSE SCHEMA: application/json

errors required — Array of object non-empty

A list of errors providing detailed information about the problem

Array [

source required — string

Enum: "source" "destination" "amount" "date" "occurrences" "frequency"

**Рис. 12.7.** Выходные данные цели

Ответ `202 Accepted` возвращается, когда перевод был принят и было предоставлено свойство `date` (для отложенных или регулярных переводов, если первый перевод был отложен). Обратите внимание, что схемы тела ответа не видны; это сделано, для того чтобы скриншот был маленьким.

В случае ошибки в запросе на перевод возвращается код состояния `400 Bad Request`. Его тело ответа показано частично (опять же, чтобы скриншот был маленьким). Однако интерес здесь представляет описание, которое дает подробную информацию обо всех возможных ошибках и то, как они представлены.

Что касается входных данных, для каждого ответа может быть предоставлено несколько примеров, как показано на рис. 12.8. Здесь выбрано `202`. Доступны образцы с отложенными и регулярными переводами. Выбран последний тип.

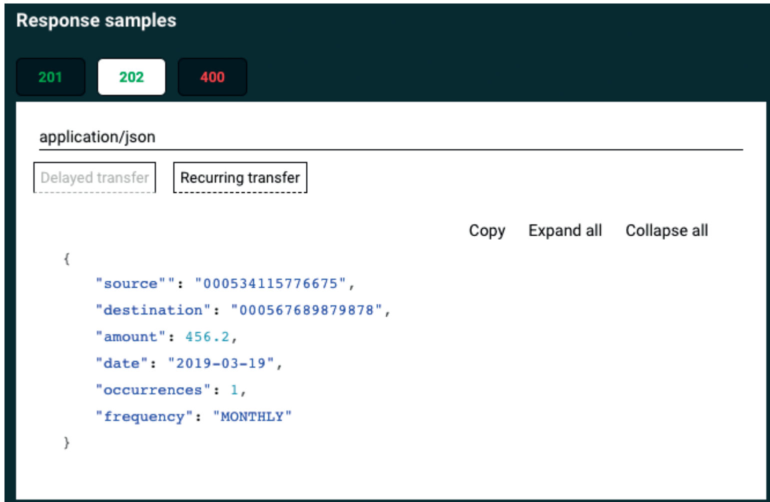



**Рис. 12.8.** Примеры выходных данных с несколькими целями

Из раздела 7.1.3 вы узнали, что, если API предоставляет несколько целей, важна их организация. На рис. 12.9 видно, как это можно сделать в справочной документации.

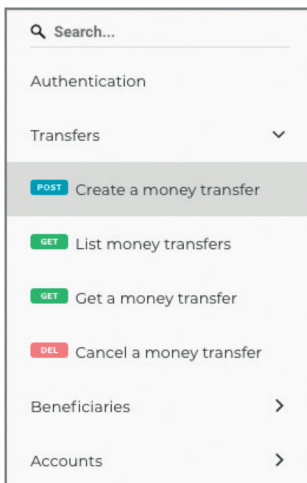



**Рис. 12.9.** Организация целей

На рис. 12.9 показано меню справочной документации (левый столбец на рис. 12.2). Цели организованы в различные категории, причем категория **Переводы** включает в себя все цели, связанные с переводами (например, «Создать денежный перевод» или «Отменить денежный перевод»). Как и в справочной документации по модели данных, все, что показано здесь, взято из базового файла спецификации OpenAPI. Приведенный ниже листинг применяет на практике то, что вы узнали в разделе 4.2.3 для описания цели.

**Листинг 12.3. Базовая справочная документация цели «Перевести деньги»**

```
[...]
paths:
  /transfers: ①
    post: ②
      summary: Creates a money transfer ③
      requestBody: ④
        content:
          "application/json":
            schema:
              $ref: "#/components/schemas/TransferRequest"
      responses: ⑤
        201:
          description: Immediate or recurring transfer
          ➥ executed
          content:
            "application/json":
              schema:
                $ref: "#/components/schemas/
                ➥ TransferResponse"
        [...]
        400:
          description: Transfer rejected ⑥
          content:
            "application/json":
              schema:
                $ref: "#/components/schemas/Error" ⑦
[...]
```

① Путь к ресурсу.
② Метод HTTP, используемый для ресурса.
③ Краткое описание пары (ресурс, метод).
④ Ввод цели.
⑤ Возможные выходные данные цели.
⑥ Краткое описание ответного сообщения.
⑦ Ссылка на модель данных ответного сообщения.

Цель описана кратко, и все возможные ответы и их модели данных перечислены без особых подробностей. Различные скриншоты справоч-

ной документации, которые вы видели, предоставили больше информации, потому что используемый файл спецификации OpenAPI, очевидно, содержит больше информации. В приведенном ниже листинге показано базовое подробное описание цели «Перевести деньги».

**Листинг 12.4. Файл спецификации OpenAPI для обзора цели «Перевести деньги»**

```
paths:
  /transfers:
    post:
      summary: Transfers money
      description: |  ①
        This operation allows one to transfer an `amount`
        of money from a `source` account to a destination`
        account.
        There are three different types of money transfer:
        - Immediate, they are executed as soon as the
        request is received
        - Delayed, they are executed upon a given future
        `date`
        - Recurring, they are executed a given
        `occurrences` number of times at a given
        `frequency`, the first occurrence being executed
        immediately or at a given `date`
```

① Многострочное описание с использованием формата разметки Markdown.

Здесь после сводки (краткого описания) предоставляется полное многострочное описание с использованием формата разметки Markdown. В приведенном ниже листинге показано, как предоставить несколько примеров тела запроса цели «Перевести деньги».

**Листинг 12.5. Несколько примеров тела запроса цели «Перевести деньги»**

```
[...]
paths:
  /transfers:
    post:
      [...]
      requestBody:
        content:
          "application/json":
            schema:
              $ref: "#/components/schemas/TransferRequest"
            examples: ①
              immediate:
                [...]
              delayed:
```

```
              [...]
            recurring: ②
              summary: Recurring transfer
              description: | The money transfer is
              ➥ executed at a given date recurringly
              value:
                source": "000534115776675"
                destination: "000567689879878"
                amount: 456.2
                date: "2019-03-19"
                occurrences: 1
                frequency: "MONTHLY"
              [...]
[...]
```

① **Несколько примеров тела запроса.**

② **Пример поставляется со сводкой, описанием и значением.**

На панели **Образцы запросов** на рис. 12.6 показаны три вкладки: **Мгновенный перевод**, **Отложенный перевод** и **Регулярный перевод**. Их содержимое получено из свойства examples, которое идет после свойства schema. Каждый пример имеет сводку (имя вкладки), описание и значение (значение вкладки). К сожалению, описание примера в ReDoc не показано.

> **ПРИМЕЧАНИЕ.** Помните, что не все инструменты могут поддерживать все функции формата описания API.

Можно предоставить несколько примеров и для других типов входных параметров (например, параметров запроса или заголовка), а также для тел ответа. Говоря об ответах, вспомните, что вы узнали в разделе 5.2.2: мы должны перечислить все возможные ошибки. Очевидно, что они должны быть задокументированы, помимо перечисления возможных кодов состояния 4XX или 5XX в справочной документации. В приведенном ниже листинге показано, как воспользоваться для этого полным многострочным описанием.

**Листинг 12.6. Подробное описание ошибки**

```
[...]
paths:
  /transfers:
    post:
      [...]
      responses
        [...]
        400:
        description: | The transfer is rejected due to an
        error in the request properties or an insufficient
        balance. Each error provides the property `source`
```

```
            of the error along with a human-readable `message`
            and its `type`:
            - MANDATORY_PROPERTY: The property indicated in
            `source` is missing
            - INVALID_FORMAT: The format of the property
            indicated in `source` is invalid
            - INVALID_VALUE: The value of the property
            indicated in `source` is invalid
            - INSUFFICIENT_BALANCE: The `amount` property is
            higher than the `source` account balance
        [...]
[...]
```

Для групп с целями здесь нет ничего нового. В разделе 7.1.3 вы узнали, как определять их, как показано в этом листинге:

**Листинг 12.7. Описание тегов**

```
[...]
tags: ①
  - name: Transfers
    description: Everything you need to transfer money
paths:
  /transfers:
    post:
      summary: Create a money transfer
      tags: ②
        - Transfers
[...]
```

① **Определение и описание необязательных тегов.**

② **Теги, которым принадлежит цель.**

Таким образом, хорошая справочная документация для цели API, такая как модель данных, требует релевантного удобного для восприятия описания и соответствующих примеров (чем больше, тем лучше). Правильное описание возможных ошибок особенно важно. Помимо предоставления несколько примеров, это в основном то, что вы должны описывать при проектировании API, поэтому этот вид справочной документации легко создать, после того как API был спроектирован.

### *12.1.3 Документирование безопасности*

Справочная документация также должна содержать информацию о безопасности, как показано на рис.х 12.10 и 12.11.

Authentication

BankingAPIScopes

| Security scheme type: | OAuth2 |
|---|---|
| implicit OAuth Flow | **Authorization URL:** https://auth.bankingcompany.com/authorize<br>**Scopes:**<br>• `transfer:create` - Create transfers<br>• `transfer:read` - Read transfers<br>• `transfer:delete` - Delete transfers<br>• `transfer:admin` - Create, read, and delete transfers<br>• `beneficiary:create` - Create beneficiaries |

**Рис. 12.10.** Как защищен банковский API и каковы доступные группы?

Create a money transfer

This operation allows one to transfer an `amount` of money from a `source` account to a `destination` account. There are three different types of money transfer:

- Immediate -- these are executed as soon as the request is received
- Delayed -- these are executed upon a given future `date`
- Recurring -- these are executed a given `occurrences` number of times at a given `frequency` -- the first occurrence being executed immediately or at a given `date`

AUTHORIZATIONS: BankingAPIScopes ( `transfer:create,` `transfer:admin` )

**Рис. 12.11.** Какие группы необходимы для создания денежного перевода

В разделе **Аутентификация** на рис. 12.10 показано, что банковский API защищен с использованием неявного потока OAuth 2.0, а (видимыми) доступными группами являются `transfer:create`, `transfer:read`, `transfer:delete`, `transfer:admin` и `beneficiary:create`. На экране **Создать денежный перевод** на рис. 12.11 в разделе **Полномочия** сказано, что потребители должны иметь группу `transfer:create` или `transfer:admin`, чтобы получить разрешение на использование цели «Перевести деньги». Здесь нет ничего нового; вы уже видели, как описывать безопасность API с помощью спецификации OpenAPI в разделе 8.2.4. В приведенном ниже листинге приведена выдержка из базового файла спецификации OpenAPI.

**Листинг 12.8.** Определение безопасности API и привязка групп к цели

```
[...]
components:
  securitySchemes: ①
    BankingAPIScopes:
      type: oauth2
```

```
          flows:
            implicit:
              authorizationUrl: "https://auth.bankingcompany.
              ➥ com/authorize"
            scopes:
              "transfer:create": Create transfers
              "transfer:read": List transfers
              [...]
[...]
  paths:
    /transfers:
      post:
        summary: Create a money transfer
        security: ②
          - BankingAPIScopes:
            - "transfer:create"
            - "transfer:admin"
[...]
```

① **Определения безопасности и группы.**
② **Группы, необходимые для использования цели.**

ReDoc автоматически добавляет меню аутентификации, показывая все, что было определено в разделе файла спецификации OpenAPI `component.securitySchemes`. Если группы `security` определяются для цели, ReDoc также показывает запись AUTHORIZATIONS:.

Опять же, поскольку вы должны определить, как защищен API при его проектировании, не требуется больших усилий, чтобы предоставить базовую справочную документацию, описывающую, как защищен API, доступные группы и какие группы необходимы для использования каждой цели.

### 12.1.4 *Обзор API*

И последнее, но не менее важное, – справочная документация на уровне API. На рис. 12.12 показано, как ReDoc использует для этого раздел файла спецификации OpenAPI info (показано в листинге 12.9).

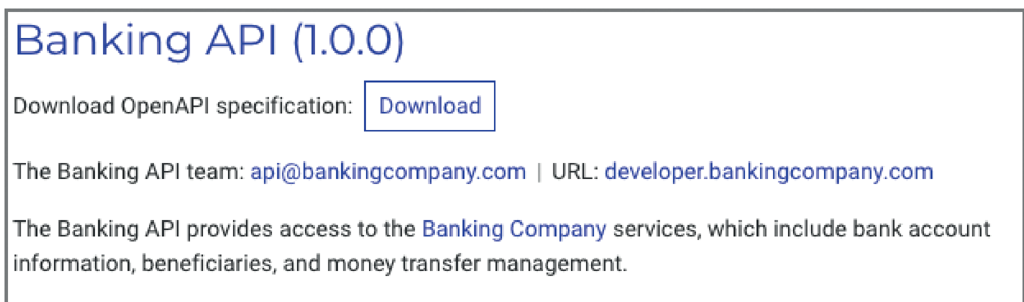



**Рис. 12.12. Краткое описание API и контактная информация в справочной документации**

**Листинг 12.9. Раздел info базового файла спецификации OpenAPI**

```
info:
  title: Banking API
  version: «1.0.0»
  description: |
    The Banking API provides access to the [Banking
    Company](http://www.bankingcompany.com) services,
    which include bank account information, beneficiaries,
    and money transfer management.
  contact:
    name: The Banking API team
    email: api@bankingcompany.com
    url: developer.bankingcompany.com
```

Вы узнали, как определить имя (`title`) и версию API при его описании с использованием спецификации OpenAPI в разделе 4.2.1. Здесь мы добавили описание и некую контактную информацию. В описании представлен обзор банковского API и используется формат Markdown для предоставления ссылки на сайт банковской компании. Контактная информация состоит из названия команды, управляющей API, их адреса электронной почты и URL-адреса сайта разработчика API.

Предоставление краткого описания API обязательно в справочной документации; это помогает потребителям понять, что можно сделать с помощью этого API, если одного его имени недостаточно. Контактная информация не является обязательной, но вы всегда должны предоставлять пользователям способ получения дополнительной информации или справки.

### 12.1.5 Генерирование документации из кода реализации: плюсы и минусы

Документацию можно сгенерировать только из кода реализации или с помощью кода плюс аннотации; таково было первоначальное намерение фреймворка Swagger (см. раздел 4.1.1).

Такая стратегия имеет преимущество, потому что поддерживает синхронизацию реализации и документирования, но учтите, что у нее есть несколько недостатков:

- я не рекомендую полагаться только на чистую генерацию на основе кода, поскольку получившаяся в итоге документация будет далеко не полной;
- существующие фреймворки аннотаций, по крайней мере, тот, с которым я работал, не позволяют использовать ту же гибкость, которую вы получаете, работая непосредственно с форматом описания API (например, невозможно предоставлять примеры, адаптированные к различным контекстам при использовании универсальных структур данных, совместно используемых в API);

- включение документации в код подразумевает, что фактически вы измените код, чтобы исправить документацию. Это может быть проблемой в зависимости от того, кто над чем работает (документация против кода) и от вашей организации, а также от вашего уровня доверия при изменении кода (да, не все организации в мире могут автоматически запускать все приложения в производство при каждой фиксации, ничего не боясь);
- на ранних этапах код должен писаться для генерации документации. Если вы помните начало этой книги, то знаете, как это может предоставить доступ к точке зрения поставщика.

Очевидно, что хранение документации вне кода также может иметь некоторые недостатки; главный из них – возможность синхронизировать документацию и код. Знайте, что в этом вопросе не существует хорошей или плохой стратегии; вы должны выбрать тот способ, который подходит вам и вашей организации.

Таким образом, каковы бы ни были средства, будучи проектировщиком API, вы можете создавать хорошую справочную документацию без особых усилий, особенно если вам нужно немного времени, чтобы предоставить подробные машиночитаемые и удобные для восприятия человеком описания и (несколько) примеров. Как и проектирование, написание хорошей документации требует практики, а справочная документация – неплохое место, чтобы начать. Помните, что могут понадобиться опытные технические писатели, если эта документация предназначена для партнерского или открытого API.

## 12.2 Создание руководства пользователя

Полные списки справочной документации и описание каждого компонента API являются обязательными, но, как упоминалось ранее, когда вы следуете рецепту, если у вас есть только список ингредиентов, вы можете изо всех сил стараться, чтобы получилось что-то съедобное. Руководство пользователя API, как и пример на рис. 12.13, предназначено для объяснения того, как на самом деле использовать API. В нем описывается, как использовать API в целом, а также его принципы и как получить к нему доступ (регистрация и получение токенов доступа). Когда вы предоставляете открытые API, эта документация может быть довольно динамичной.

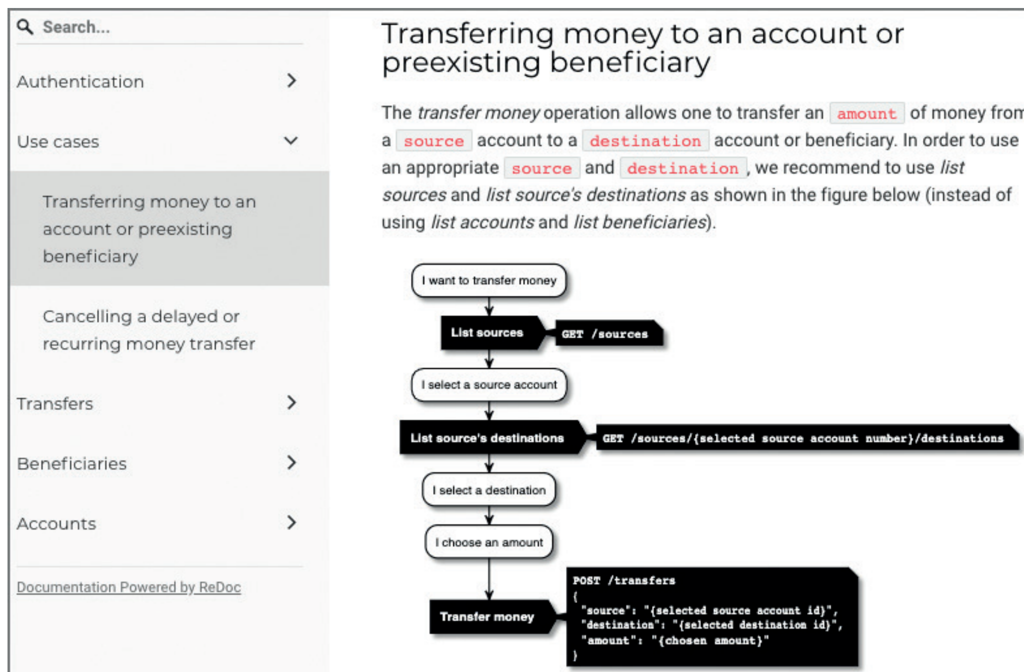



**Рис. 12.13.** Руководство пользователя API

## 12.2.1 Документирование вариантов использования

На рис. 12.13 в левой панели есть меню **Варианты использования**. В нем содержится два элемента: «Перевод денег на счет или уже существующему бенефициару» и «Отмена отложенного или регулярного перевода». Выбран первый вариант. Очевидно, что каждый элемент предназначен для описания варианта использования, который показывает, как можно сочетать различные цели API.

В правой панели, где показан вариант использования «Перевода денег», есть текст, объясняющий, что такое перевод денег, концепцию исходных и целевых счетов и какие цели следует использовать, чтобы выбрать соответствующие значения для этих свойств. Ниже идет диаграмма, на которой перечислен порядок действий:

1. Вызовите цель «Перечислить исходные счета».
2. Выберите исходный счет из получившегося списка.
3. Вызовите цель «Перечислить счета назначения» для выбранного источника.
4. Выберите счет назначения из возвращенного списка.
5. Определитесь с суммой.
6. Вызовите цель «Перевести деньги», указав выбранный исходный счет и счет назначения, а также сумму.

Вам это не кажется знакомым? Похоже на информацию, которую вы научились собирать в разделе 2.3 и помещать ее в таблицу целей API, как показано на рис. 12.14.

| Участники | Действия | Способы совершения действий | Входные данные (источник) | Результаты (использование) | Цели |
|---|---|---|---|---|---|
| Клиент | Перевод денег | Выбрать исходный счет | Идентификатор клиента (защищенные данные) | Исходные счета, используемые для перевода денег (перечислить счета назначения, перевод денег) | Перечислить исходные счета |
| | | Выбрать счет назначения или получателя | Исходный счет (перечислить исходные счета) | Действительные счета назначения или предварительно зарегистрированные бенефициары для выбранного исходного счета (перевести деньги) | Перечислить счета назначения |
| | | Запрос на перевод денег | Исходный счет (перечислить исходные счета), счет назначения или предварительно зарегистрированный бенефициар (перечислить счета назначения) и сумма (ввод со стороны клиента / потребителя) | Идентификатор денежного перевода | Перевести деньги |

**Рис. 12.14.** Отрывок из таблицы целей API, созданный при проектировании банковского API.

Таблица целей API описывает варианты использования, которых можно достичь с использованием различных целей. Это означает, что работу, выполненную во время проектирования API, можно использовать повторно для документирования API. Таблицу целей API можно более или менее использовать повторно как для закрытых API, так и в качестве необработанных входных данных для партнерских или открытых API.

Описать варианты использования в руководстве пользователя API можно разными способами, используя множество различных инструментов. В случае с базовыми руководствами все, что вам нужно, – это уметь писать форматированный текст и, возможно, добавлять какие-нибудь диаграммы или изображения. Вы можете использовать простую систему управления контентом (CMS), портал для разработчиков (который обычно включает в себя функции CMS) или даже создать собственный веб-сайт. В целях иллюстрации я буду продолжать использовать ReDoc и файл спецификации OpenAPI, как показано в приведенном ниже листинге.

**Листинг 12.10. Как ReDoc использует преимущества файла спецификации OpenAPI**

```
[...]
info:
  title: Banking API
  version: «1.0.0»
  description: |  ①
    The Banking API provides access to the [Banking
    Company]  (http://www.bankingcompany.com) services,
    which include bank account information, beneficiaries,
    and money transfer management.
    # Use cases ②
```

```
    ## Transferring money to an account or preexisting
      ➥ beneficiary ③
    The _transfer money_ operation allows one to transfer
    an `amount` of money from a `source` account to
    a `destination` account or beneficiary.
    In order to use an appropriate `source` and
    `destination`, we recommend to use _list sources_ and
    _list source's destinations_ as shown in the figure
    below (instead of using _list accounts_ and _list
    beneficiaries_).
    ![Diagram](http://developer.bankingcompany.com/
    ➥ diagrams/transfer.svg) ④
    ## Canceling a delayed or recurring money transfer ⑤
    - List money transfers: To list existing money
      transfers and select the one to delete
    - Cancel a money transfer: To cancel the selected
      money transfer
[...]
```

① **Описание API.**
② **Заголовок первого уровня Markdown.**
③ **Заголовок второго уровня Markdown.**
④ **Это описание включает в себя изображение.**
⑤ **Простое текстовое описание.**

Я добавил текст и поместил изображение в раздел `info.description` файла спецификации OpenAPI, используя преимущества формата Markdown. Если описание содержит заголовки первого и второго уровней, ReDoc автоматически добавляет их в левую панель в качестве пунктов меню и подменю. Если оно содержит изображение (`![Text](URL)`), оно отображается, пока браузер может получить доступ к URL-адресу, в котором работает документация ReDoc.

Безусловно, диаграммы могут оказать большую помощь потребителям (как говорится, одна картинка стоит тысячи слов), но вы также можете просто использовать текст, как показано во втором случае использования (отмена отложенного или регулярного денежного перевода).

## Код вашей диаграммы

Если вы хотите создавать диаграммы, но не работаете с инструментами для рисования, обратите внимание, что диаграмма, показанная на рис. 12.13, была сгенерирована с использованием инструмента PlantUML (или иногда PUML), который доступен по адресу http://plantuml.com. Этот инструмент позволяет создавать диаграммы с использованием кода. Код диаграммы перевода денег можно найти в исходном коде для этой книги (https://www.manning.com/books/the-design-of-web-apis). Изучите веб-сайт инструмента, чтобы узнать, как использовать этот удивительный формат.

В случае с большими API со множеством вариантов использования в долгосрочной перспективе включением руководства пользователя в файл описания API может быть сложно управлять. Ответственные за поддержку описания API могут не заниматься поддержкой руководства пользователя, а у двух форм документации может быть разный жизненный цикл. Если вы используете совершенно другой инструмент или систему для руководства пользователя API, не забудьте взять `info.contact.url` или добавить ссылку в описании, указывающую на то, где оно находится.

### 12.2.2 Документирование безопасности

Еще одна важная тема, касающаяся руководства пользователя API, – это то, что потребители должны делать, чтобы вызвать API. Было бы жаль, если бы потребители знали, какие API-вызовы следует совершать, чтобы инициировать перевод денег, но на самом деле не могли бы делать эти вызовы, потому что они не знают, как получить токен безопасности. Руководство пользователя API *должно* включать в себя советы касательно того, как зарегистрироваться в качестве разработчика, зарегистрировать потребительское приложение и получить токены, используя доступные потоки OAuth или любую другую систему или фреймворк (см. раздел 8.1).

К сожалению, на этапе проектирования здесь нет ничего такого, что можно было бы использовать повторно (список групп не очень помогает); но, к счастью, такая документация должна быть практически одинаковой для всех ваших API, и существуют буквально тысячи ресурсов, которые можно использовать для объяснения того, как получить токен доступа с помощью потока OAuth после регистрации потребительского приложения.

### 12.2.3 Предоставление обзора общепринятого поведения и принципов

Руководство пользователя API также может содержать информацию обо всех общепринятых вариантах поведения и принципах API; безопасность – лишь один из них. Такая документация может объяснить, как обрабатываются ошибки (см. раздел 5.2.3), доступные форматы данных и поддерживаемые языки или как обрабатывается нумерация страниц (см. раздел 6.2). В общем, вы должны включить все, что является общепринятым для ваших целей API, и стоит упомянуть потребителей, чтобы облегчить использование API.

### 12.2.4 Мышление вне статической документации

Эта тема полностью выходит за рамки этой книги, но знайте, что статическая документация не единственный вариант. Удостоившиеся самой высокой оценки (открытые) API-интерфейсы обычно предоставляют отличные порталы для разработчиков, в том числе высококачественную справочную документацию и руководства пользователя, – и все это построено абсолютно динамично.

Например, просматривая справочную документацию такого API, можно увидеть кнопку **Попробовать!**, которая позволяет вызывать API с помощью предварительно заполненного запроса, а портал разработчика обрабатывает все вопросы, связанные с безопасностью «под капотом». Аналогичным образом некоторые руководства пользователя позволяют пошагово тестировать варианты использования на портале для разработчиков[1].

## *12.3 Предоставление адекватной информации разработчикам*

Мы изучили, как документировать API для потребителей, но, прежде чем кто-либо из них начнет использовать API, его необходимо реализовать. А что нужно людям, ответственным за реализацию? Очевидно, что им необходимо подробное описание контракта интерфейса и, возможно, того, как он должен использоваться; но этого недостаточно. Им также нужно описание того, что происходит «под капотом».

> **ПРИМЕЧАНИЕ.** События, описанные в следующем рассказе, являются вымышленными. Любое сходство с любым живым или умершим человеком или любой существующей или предыдущей организацией просто случайно.

Когда банковский API был реализован, полученный API оказался не совсем таким, как ожидалось. Например, остатки на счетах были реализованы в качестве объектов, содержащих значение в виде числа и валюту в виде строки. К сожалению, значение баланса в размере 123,45 долл. США выглядело так: `12345` (остаток в центах, как он хранится в банковской системе), а значение валюты так: `"C123"` (внутренний код валюты $). Кроме того, возвращаемые балансы были не балансами в реальном времени, а ежедневными балансами, которые обновлялись один раз в день – в полночь.

Также были проблемы и с обработкой ошибок. Например, когда в запросе на перевод денег отсутствовал и счет назначения, и сумма, в сообщении об ошибке указывалось только на первую проблему. А если остаток на счете был недостаточным, тип ошибки представлял собой внутренний код ошибки `"0002"`. Еще страшнее, если потребитель, используя токен безопасности, связанный с данным клиентом, запрашивал существующий счет с помощью запроса `GET /accounts/{accountNumber}` и он не принадлежал потребителю, потребитель получал ответ `200 OK` и счет вместо ошибки `404 Not Found`. К счастью, у кого-то возникла идея провести тесты, которые выявили эти проблемы (мы поговорим об этом подробнее в разделе 13.3.5).

Проведя расследование, команда проекта поняла, что разработчики, отвечающие за реализацию, не получили достаточно информации о контракте API, о том, как он должен отображаться в базовой системе и каковы ожидаемые меры безопасности. Что еще более важно, им не

---

[1] Обратите внимание на сайты платформы Twilio (https://www.twilio.com) и компании Stripe (https://stripe.com), где можно найти высококачественную документацию.

хватало знаний о безопасности API в целом. Все проблемы были решены путем улучшения описания API (файла спецификации OpenAPI), как показано на рис. 12.15 и в листинге 12.11, а также путем обучения и руководства.

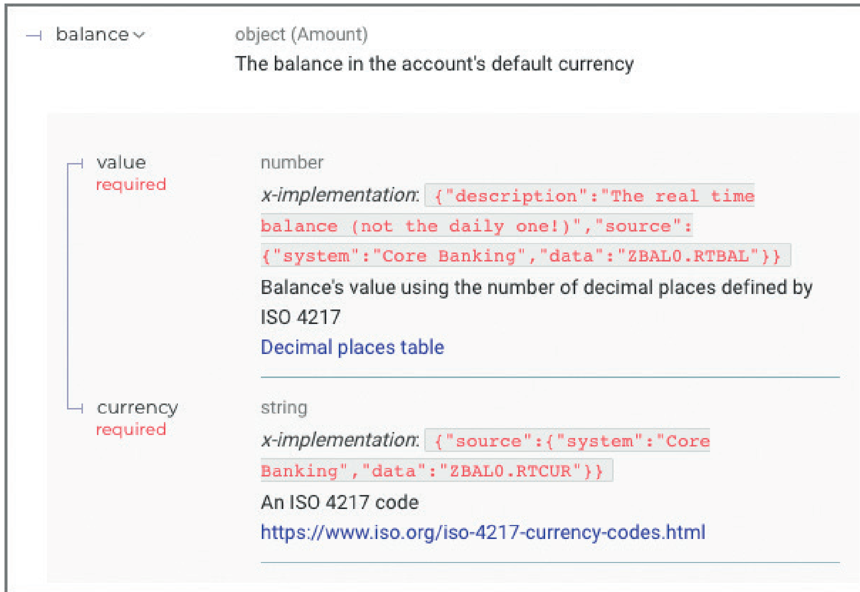



**Рис. 12.15.** Описание API с расширенным описанием и информацией о реализации

**Листинг 12.11. Добавление пользовательских свойств в файл спецификации OpenAPI**

```
[...]
properties:
  value:
    description: |
      Balance's value using the number of decimal places
      as defined by ISO 4217
    externalDocs: ①
      description: Decimal places table
      url: https://www.currency-iso.org/en/home/tables/
      ➥ table-a1.html
      type: number
      x-implementation: ②
        description: The real time balance (not the daily
          one!)
        source:
          system: Core Banking
          location: ZBAL0.RTBAL
    currency:
      description: An ISO 4217 code
```

```
        externalDocs: ③
          url: https://www.iso.org/iso-4217-currency-codes.html
        type: string
        example: USD
        x-implementation:
          source:
            system: Core Banking
            location: ZBAL0.RTCUR
[...]
```

① **Ссылка на документацию.**
② **Пользовательские данные игнорируются стандартными парсерами.**
③ **Ссылка без описания.**

Проблемы с форматом данных были вызваны неполным описанием модели данных суммы в файле описания API. Если бы в описании свойства значения суммы было указано, что количество знаков после запятой определялось стандартом ISO 4217, это дало бы подсказку относительно его формата. То же самое относится и к описанию валюты, в котором должно быть указано, что это значение представляет собой трехбуквенный код согласно стандарту ISO 4217. Для обоих свойств были добавлены ссылки на внешнюю документацию (`externalDocs`) о ISO 4217. Обратите внимание, что первая ссылка на таблицу десятичных разрядов содержит описание, которое ReDoc использует для визуализации ссылки.

Что касается информации о балансе, поскольку банковский API используется неопытными потребителями, не было необходимости указывать в его описании, что это баланс в реальном времени. Это означает, что это должно быть указано в другом месте, которое можно увидеть только на стороне поставщика. Команда (включая проектировщика и разработчиков) решила определить его в файле спецификации OpenAPI с помощью пользовательского расширения `x-implementation`, в котором содержалось подробное описание источника данных (системы и местоположения).

Стандартные парсеры спецификации OpenAPI игнорируют все свойства, начинающиеся с `x-`, поэтому имя и формат свойства `x-implementation` полностью настраиваются и определялись командой. Такие инструменты, как ReDoc, могут отображать данные, содержащиеся в таком свойстве, но, как видно на рис. 12.15, они отображаются как объект JSON. Для более удобной визуализации нужно настроить инструмент. Файл спецификации OpenAPI также был расширен, чтобы в нем содержалась информация о мерах безопасности, как показано в приведенном ниже листинге.

**Листинг 12.12. Безопасность контролирует информацию о цели «Получить счет»**

```
[...]
paths:
  /accounts/{id}:
    get:
```

```
        summary: Get an account
        x-implementation:
          security:
            description: | Only accounts belonging to user
            referenced in security data;
            return a 404 if this is not the case
          source:
            system: security
            location: jwt.sub
          fail: 404
[...]
```

Свойство `x-implementation` содержит описание элементов управления безопасностью, которые необходимо выполнить, данные безопасности, которые нужно использовать, и какой статус HTTP нужно вернуть в случае сбоя. Помимо этой подробной документации по реализации, разработчиков обучали разбираться в проблемах безопасности API, а также им были предоставлены методические руководства (мы поговорим об этом в разделе 12.2). Проблемы с ошибками были исправлены путем добавления более подробной информации в документацию, ориентированную на потребителя (как показано в разделе 12.1.1), а также путем объяснения принципов обработки ошибок (как видно из раздела 12.1.2).

**ПРИМЕЧАНИЕ.** Документация, ориентированная на потребителя, не должна содержать информацию о `x-implementation`; ее нужно удалить из файла спецификации OpenAPI перед публикацией на портале для разработчиков.

Итак, первым шагом на пути к предоставлению соответствующей документации для разработчиков API является предоставление подробной документации для потребителей (справочная документация и руководство пользователя). Эта документация не должна быть броской и привлекательной, но должна быть исчерпывающий. Но этого недостаточно.

Разработчикам также нужна *ориентированная на поставщика* документация о том, что на самом деле происходит «под капотом». Им нужна информация об отображении данных (из какой системы поступает каждый фрагмент данных), отображении ошибок (как преобразовать внутренние ошибки в ошибки, с которыми сталкиваются потребители), данных о безопасности и средствах управления, а также ожидаемом поведении, основанном на внутренних бизнес- или технических правилах.

Как вы уже видели, эту информацию можно задокументировать в файле описания API, но это только один из вариантов. Выбор способа предоставления этой информации зависит от вас, как от проектировщика API, и от людей, с которыми вы работаете. Помимо актуальной документации API, документация разработчиков также состоит из руководства и обучения (см. раздел 12.2).

## 12.4 Документирование изменений API и устаревшие функции

В разделе 9.1 вы узнали, как обращаться с проектированием и изменениями API, чтобы ограничить внесение критических изменений. Но изменения, критические или нет, неизбежно произойдут, и их нужно задокументировать.

Такая документация полезна для потребителей, чтобы держать их в курсе новых возможностей и сообщать, нужно ли им менять свой код в случае, если элементы устарели (или, что еще хуже, уже были удалены). Это может быть также полезно для всех тех, кто участвует в проекте, предоставляя им обзор предстоящих изменений в следующей версии.

И кто лучше всего документирует или хотя бы перечисляет все эти изменения? Вы! Как человек, который спроектировал эти изменения, вы лучше всех знаете, что вы сделали. На рис. 12.16 показан очень простой *журнал изменений*, описывающий последние изменения, сделанные в банковском API.

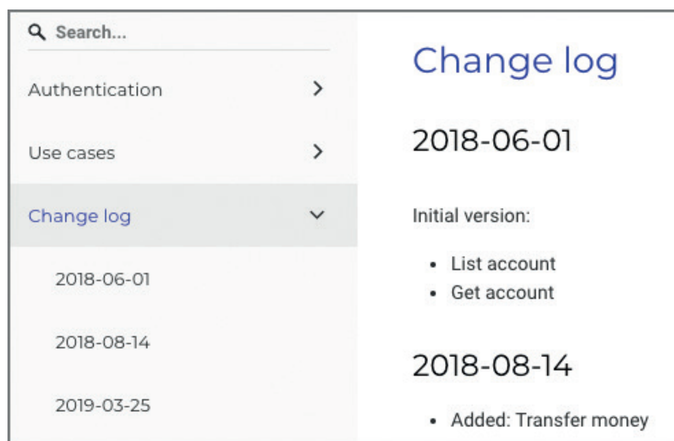



**Рис. 12.16.** Простой журнал изменений, в котором перечислены изменения, сделанные в каждой версии

В журнале изменений должно быть указано, какие элементы (свойства модели данных, параметры, ответы, группы безопасности) были добавлены, изменены, объявлены устаревшими или удалены. Здесь мы просто воспользуемся преимуществом раздела `info.description` файла спецификации OpenAPI, чтобы добавить заголовок журнала изменений первого уровня, содержащий разделы второго уровня для каждой версии, как мы это делали для вариантов использования. Форматы описания API, по крайней мере, на момент написания этой книги, не предлагают способов описания такого журнала изменений, но они могут хотя бы предоставить способы указания устаревших элементов, как показано на рис. 12.17.

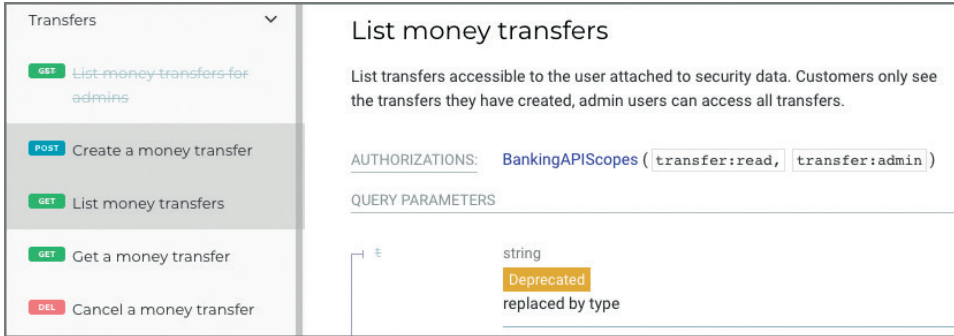



**Рис. 12.17.** Указание устаревших элементов с использованием файла спецификации OpenAPI

Цель «Перечислить денежные переводы для администраторов» в левом меню зачеркнута, а параметр `t` запроса цели «Перечислить денежные переводы», показанный в правой панели, обозначен как устаревший; и там и там флаг `deprecated` установлен в значение `true` в файле спецификации OpenAPI, как показано в листингах 12.13 и 12.14.

**Листинг 12.13.** Устаревшая цель «Перечислить денежные переводы для администраторов»

```
/admin-transfers:
   get:
      summary: List money transfers for admins
      tags:
         - Transfers
      description: Redirects to GET /transfers
      deprecated: true ①
   responses:
      "200":
         description: Transfers list
         content:
            "application/json":
               schema:
                  $ref: "#/components/schemas/TransferList"
```

① Флаг, указывающий на то, что цель устарела.

**Листинг 12.14.** Устаревший параметр запроса t

```
/transfers:
   get:
      summary: List money transfers
      parameters:
      - name: t
         in: query
         description: replaced by type
         deprecated: true ①
```

```
        schema:
          type: string
      - name: type
          in: query
          description: transfer type
          schema:
            type: string
```

① **Флаг указывающий на то, что параметр устарел.**

В соответствии со спецификацией OpenAPI флаг `deprecated` можно использовать для параметров, целей и свойств в моделях данных. Здесь описания устаревших элементов указывают на то, что использовать в качестве замены. Эти описания также могут указывать на то, когда устаревшие элементы будут удалены (если это должно быть сделано). Вы также можете применить пользовательские свойства `x-` для добавления дополнительной структурированной информации об депрекациях, вместо того чтобы использовать текст в описаниях.

Документирование устаревших элементов иногда можно выполнять динамически путем предоставления метаданных в ответах API. Например, заголовок Sunset, определенный в RFC 8594 (https://tools.ietf.org/html/rfc8594), позволяет серверу сообщать о том факте, что ресурс может перестать отвечать в определенный момент времени. Если банковская компания представляет версию 2 банковского API 4 августа 2019 года и позволит всем потребителям в течение шести месяцев обновлять свой код, любой вызов, выполненный на любом ресурсе версии 1 (например, запрос `GET /v1/accounts`), может вернуть ответ, показанный в приведенном ниже листинге, заявив, что ресурс не будет доступен после 4 февраля 2020 года.

**Листинг 12.15. Ответ с заголовком Sunset**

```
200 OK
Sunset: Tue, 4 Feb 2020 23:59:59 GMT
{ «items»: [ ... ] }
```

В следующей и последней главе мы еще больше расширим контекст проектирования API, чтобы иметь возможность работать в качестве разработчиков API со множеством API на долгосрочной основе, даже с теми, которые мы не проектируем.

## *Резюме*

- Проектировщики API должны участвовать в создании различных типов документации API.
- Подробная справочная документация – это хорошо, но этого недостаточно. Также нужно создать руководство пользователя.
- Руководства пользователя должны содержать всю необходимую информацию для использования API в целом, включая способы получения учетных данных и токенов.

- Использование языка описания API, такого как спецификация OpenAPI, может помочь при создании документации.
- Важно отслеживать изменения, чтобы информировать пользователей об изменениях.
- Создание документации помогает протестировать дизайн API.

13

# Развитие API

**В этой главе мы рассмотрим:**

- жизненный цикл API;
- рекомендации по проектированию API;
- обзоры API;
- общение и сообщество.

В ходе прочтения этой книги мы расширили наше видение проектирования API и API-интерфейсов в целом, выйдя за пределы простых программных интерфейсов приложения. Это особенно верно в отношении четырех предыдущих глав. В главах 10 и 11 вы видели, как обращать внимание не только на контекст потребителя, но и на контекст поставщика, чтобы проектировать реалистичные API, которые будут реализуемыми и будут удовлетворять потребности потребителей наиболее эффективным способом. И, не ограничиваясь текущей версией API, в главе 9 вы научились создавать конструкции API, которые снижают риск внесения критических изменений при их модификации.

Проектирование API требует о нас, чтобы мы думали не только о самих API, потому что они являются лишь одной частью целого. Мы начали раскрывать это целое в главе 12. Вы обнаружили, что проектировщики API могут делать гораздо больше, чем просто создавать API, участвуя в создании различных видов документации. В этой последней главе мы изучим другие три темы, связанные с проектированием API, о которых должен знать любой проектировщик API, а иногда и осваивать их, чтобы в конечном итоге расширить свои API.

Сначала мы поговорим о жизненном цикле API: как API рождаются, живут и в конце концов становятся ненужными. Затем мы изучим рекомендации по API, которые являются обязательными при проектировании нескольких API или нескольких версий одного API, независимо от того, работаете ли вы в одиночку или с другими специалистами. После этого мы рассмотрим различные способы проверки API, чтобы убедиться, что дизайн соответствует поверхности API для организации, отвечает ожиданиям, реализуем и удовлетворяет потребителей. В заключение мы поговорим об общении и обмене вашими API, их изменениях и практиках использования API.

## 13.1 Жизненный цикл API

Развитие API требует, чтобы мы знали жизненный цикл API и понимали, что он работает параллельно с другими, как показано на рис. 13.1. Как видите, проектирование API – это только часть жизненного цикла API: как API рождаются, живут и в конце концов становятся ненужными.

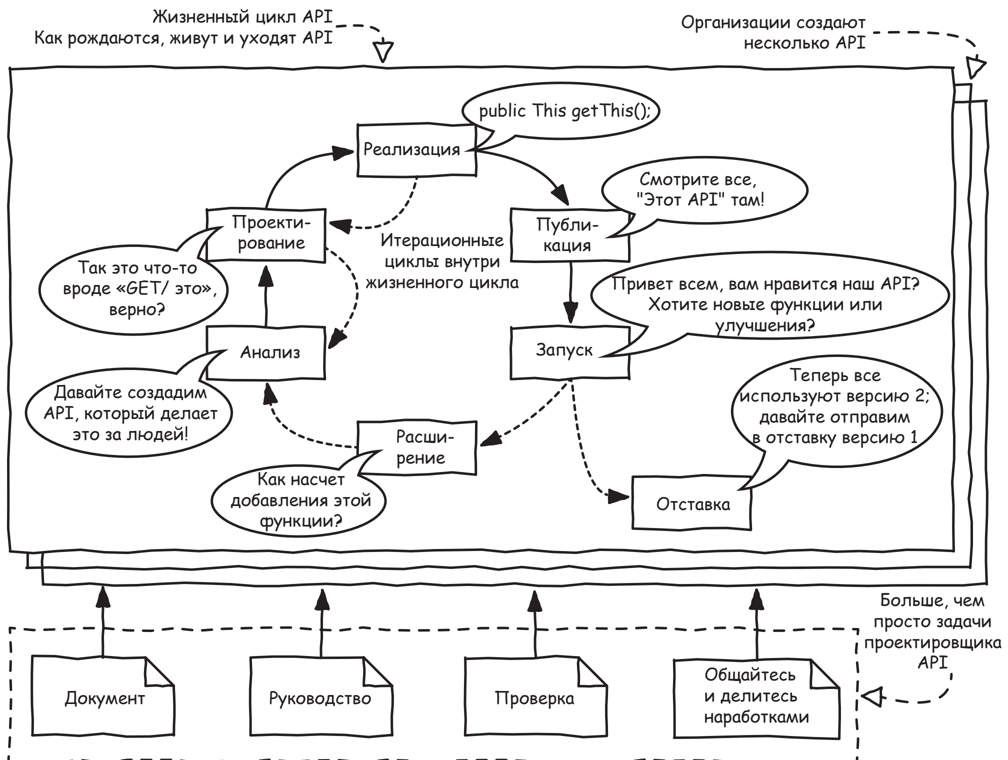



Рис. 13.1. Жизненный цикл API

Жизненный цикл API начинается с *фазы анализа*, когда компания/организация/команда/человек думают, что было бы интересно предоставить API для бизнеса или по техническим причинам. На этом этапе рассматриваются такие темы, как цели API, какие потребности он должен

выполнять, на каких потребителей он ориентирован, кому он нужен и какие преимущества он предлагает. Затем наступает *этап проектирования*, на котором идеи, возникающие на этапе анализа, основательно исследуются и транскрибируются в контракт интерфейса. После этого, на *этапе реализации*, создается приложение, предоставляющее доступ к этому контракту.

Анализ, проектирование и реализация на самом деле являются итерационным процессом; вам, возможно, придется двигаться вперед и назад в соответствии с новыми открытиями, новыми вопросами или просто потому, что вы передумали и не хотите использовать конкретное дизайнерское решение. После этого API становится доступным для целевых потребителей на *этапе публикации*. API будет работать, до тех пор пока не будет развиваться, предоставляя новые функции. Но от него также могут избавиться, потому что необходима другая версия, вводящая критические изменения, с заменой предыдущей на новую. К сожалению, от API также могут и избавляться, потому что они оказались неудачными или просто больше не нужны.

Для детального изучения полного жизненного цикла API понадобилась бы целая книга, если не несколько. В оставшейся части этой главы мы сосредоточимся на темах, относящихся к проектированию API и проектировщикам, но эти темы не ограничиваются фазой проектирования. Чаще всего проектировщики API должны вмешиваться в различные этапы жизненного цикла API и не просто проектировать его.

Чтобы гарантировать успех, проектировщики должны тесно сотрудничать с заинтересованными сторонами, владельцами продуктов, техническими писателями, разработчиками или тестировщиками. Они также должны тесно сотрудничать с потребителями – либо напрямую, либо с помощью команды по связям с разработчиками API. А при работе над API проектировщики участвуют в создании различных типов документации (как вы видели в предыдущей главе), принимают участие в различных обзорах как на стороне поставщика, так и на стороне потребителя, а также могут участвовать в обмене информацией об API.

Но организации редко создают единый неизменный API; вместо этого они обычно создают несколько постоянно развивающихся API. Проектировщики API должны работать вместе, чтобы создать согласованную поверхность API для организации (см. раздел 6.1). Они должны делиться тем, что они делают, помогая друг другу, предоставляя рекомендации при проектировании и обзоре API. Даже отдельные проектировщики API должны сами направлять себя, следя за тем, что уже было сделано, чтобы быть последовательными.

## 13.2 Создание руководства по проектированию API

Посадите двух проектировщиков API в конференц-зале и спросите их, как обрабатывать нумерацию страниц в REST API. Вы, вероятно, станете свидетелями оживленной дискуссии, в которой каждый предложит разные методы и оценит все за и против решений, предложенных собесед-

ником. В конце вам может быть предложено несколько вариантов. Все они рабочие, и некоторые из них отличаются от тех, что используются в вашей существующей организации или API-интерфейсах вашей команды, которые были созданы теми же специалистами либо другими.

В разделе 6.1 вы видели, что согласованность является ключевой проблемой при проектировании API. API должны быть согласованы с самими собой, с остальными API организации/команды, а также с окружающим миром, чтобы быть простыми в использовании. Быть согласованным во времени, работая в одиночку над одним API – проще сказать, чем сделать, но, когда большое количество проектировщиков работают над поверхностью API, которая состоит из множества API, трудно поддерживать эту поверхность абсолютно гладкой. Мы – люди; у всех нас есть свои предпочтения и предпосылки, а иногда мы меняем свое мнение, даже не замечая этого.

Когда приходят новые проектировщики, они должны выяснить, как нужно проектировать ваши API, чтобы быть последовательными. Но без надлежащего руководства они вряд ли добьются успеха. Точно так же, если люди, никогда ранее не занимавшиеся проектированием API, хотели бы сделать это, было бы разумно дать им адекватные рекомендации, чтобы избежать катастроф при проектировании API.

Определение *руководства*, набора правил, которые будут использоваться всеми проектировщиками, является обязательным условием для обеспечения согласованности внутри и по всему API организации или команды. Это также хороший способ, чтобы не тратить время на бесконечные дебаты, где все правы, но нужно выбрать единственное решение и сосредоточиться на том, что действительно важно: предоставить самые простые для понимания и легкие в использовании представления API, отвечающие потребностям потребителей. Более того, руководство – прекрасный инструмент, который поможет начинающим проектировщикам API. Давайте посмотрим, как может выглядеть его содержание и как создать его.

### 13.2.1 Что включить в руководство по проектированию API

Руководство по проектированию API может состоять из трех разных уровней: *справочное руководство*, которое сосредоточено на описании основ конструкций API, *руководство по применению*, объясняющее, как применять эти основы в различных случаях использования, и *рекомендации по процессу проектирования*, которые предоставляют руководство по проектированию API. Такое руководство также может предоставлять информацию, выходящую за рамки контракта интерфейса, такую как сведения об архитектуре программного обеспечения и принципах реализации. Давайте внимательнее рассмотрим каждый уровень, что он в нем содержится и почему стоит подумать о том, чтобы включить это в руководство своего API.

#### Справочное руководство

Чтобы быть последовательным в проектировании API, необходимо определить принципы и правила, которые следует применять. Справочное

руководство – это минимальные принципы проектирования API, которые вы должны создать: в них перечисляются и описываются все принципы и правила. На рис. 13.2 приведена выдержка из справочного руководства банковской компании.

https://wiki.bankingcompany.intranet/api-guidelines/principles/

Добро пожаловать
Принципы
Случаи использования
Как проектировать API
Особенности реализации

**Принципы проектирования API**

**Определения**

*Ресурс «Коллекция»*

Ресурс «Коллекция», или просто коллекция, представляет собой список или набор единичных ресурсов. В пути он должен быть представлен именем во множественном числе. Например, /beneficiaries - это список бенефициаров.

*Единичный ресурс*

Единичный ресурс, или просто ресурс, является элементом в коллекции. Он определяется уникальным идентификатором внутри коллекции; например, /beneficiaries123456 соответствует бенефициару, чей идентификатор - 12345.

**HTTP- заголовки**

*Location*

Заголовок Location должен быть возвращен при создании ресурса с использованием HTTP-метода POST. Он будет содержать путь к созданному ресурсу. Например, для запроса POST /beneficiaries должен быть возвращен заголовок Location: /beneficiaries/{beneficiaryId} (beneficiaryId является идентификатором созданного бенефициара).

**Рис. 13.2.** Принципы проектирования API банковской компании

В самом базовом справочном руководстве может, например, описываться, какие методы HTTP, коды состояния или заголовки можно использовать и когда; формат путей к ресурсам; какие форматы данных возвращаются в случае ошибок; и как справиться с нумерацией страниц. Такие руководства также должны содержать четкие и общие определения словарного запаса, используемого при проектировании API; например, что такое API, что такое ресурс или коллекция, что такое путь и что такое версия? Если вы слышали о предметно-ориентированном проектировании, это можно сравнить с вездесущим языком, который должен использоваться всеми людьми, участвующими в проектировании API.

Справочное руководство можно сравнить со справочной документацией по API: в нем перечислены и описаны все элементы, необходимые для проектирования API. Как и справочная документация, по отдельности они могут довольно легко усваиваться, и может быть не так просто разрабатывать API, которые соблюдают их. Вот почему вы должны также рассмотреть вопрос о добавлении руководства по случаям использования.

### Руководство по случаям использования

К настоящему времени вы должны знать, что юзабилити и точка зрения пользователя важны при проектировании. Вы были свидетелями этого при обучении проектированию API, а также их документировании. Очевидно, что рекомендации по проектированию API не являются исключением: вы должны создавать их с учетом удобства и простоты использо-

вания. Если вы этого не сделаете, лучше вообще не писать их, потому что в лучшем случае некоторые люди могут их читать, но не полностью следовать им; а в худшем случае никто вообще не захочет читать их! В руководстве по случаям использования приводятся готовые к использованию «рецепты» или решения, как показано на рис. 13.3.

На странице **Создать элемент** описывается вариант применения с использованием обычного словаря и предоставлением вариантов типичного создания элемента, а затем объясняется, как сделать это в REST API с использованием общего словаря. Описывается, какие ожидаются параметры, их формат и какой тип ответного сообщения нужно предоставить с указанием только необходимых правил или принципов, вытекающих из справочного руководства.

https://wiki.bankingcompany.intranet/api-guidelines/use-cases/

Добро пожаловать
Принципы
Случаи использования
Как проектировать API
Особенности реализации

Проектирование вариантов использования

**Создать элемент**

*Когда использовать этот вариант*

Когда вам нужно создать, добавить, запустить, сохранить, отправить, зарегистрировать ... элемент. Например, отправить электронное письмо клиенту, зарегистрировать нового получателя или перевести деньги.

*Как это делается в REST API*

Для этого добавьте единичный ресурс в коллекцию, используя запрос POST. Например используйте запрос `POST /customers/{customerId}/emails`, чтобы отправить электронное письмо клиенту или запрос `POST /transfers` для перевода денег.

*Параметры*

Разрешается только параметр тела. Данные соответствуют представлению создания/обновления/замены (см. «Модели данных» в разделе «Принципы»).

*Успешный результат*

Если действие выполнено мгновенно, будет возвращено состояние `201 Created` и `202 Accepted` если оно будет выполнено позже. Заголовок Location должен быть возвращен в обоих случаях.

*Errors*

**Рис. 13.3.** Описание варианта использования в руководстве по проектированию API банковской компании

Такая документация действительно важна для начинающих, но она также полезна для опытных проектировщиков API. Некоторые проектировщики могут время от времени выполнять только эту работу, и им может быть неудобно использовать словарь технических проектировщиков API. Некоторые из них могут придерживаться функций CRUD, и им может быть не просто проектировать цель, непохожую на создание *xyz*; обычно они испытывают желание использовать запрос типа `POST /сделать-это`. Они также могут не знать, какой статус HTTP выбрать, если в справочном руководстве содержится только перечень авторизованных кодов состояния HTTP без подробной информации о том, когда именно их использовать. Если они не прочитали все принципы, они могли упустить тот факт, что при создании ресурса должен возвращаться заголовок `Location` (даже опытные проектировщики API могут забыть об этом!).

Как начинающие, так и опытные проектировщики будут работать более эффективно и станут счастливее, если смогут найти всю необходимую информацию на одной странице, не просматривая все принципы и правила. Но как они узнали, что им нужно добавить цель «Создать *что-то*» в API?

### Руководство по процессу проектирования

Проектирование API требует методов, инструментов и процессов. Может быть полезным добавить руководство по процессу проектирования в свое руководство по API, как показано на рис. 13.4.

Идея состоит не в том, чтобы копировать и вставлять текст из этой книги или какой-либо другой книги по проектированию программного обеспечения в свое руководство. Такие рекомендации могут просто обеспечить основу для проектирования или ссылки на существующие документы или контрольные списки или учебные занятия, проводимые опытными проектировщиками API. В главах 10 и 11 вы узнали, что проектирование API – это не просто вопрос проектирования контракта интерфейса.

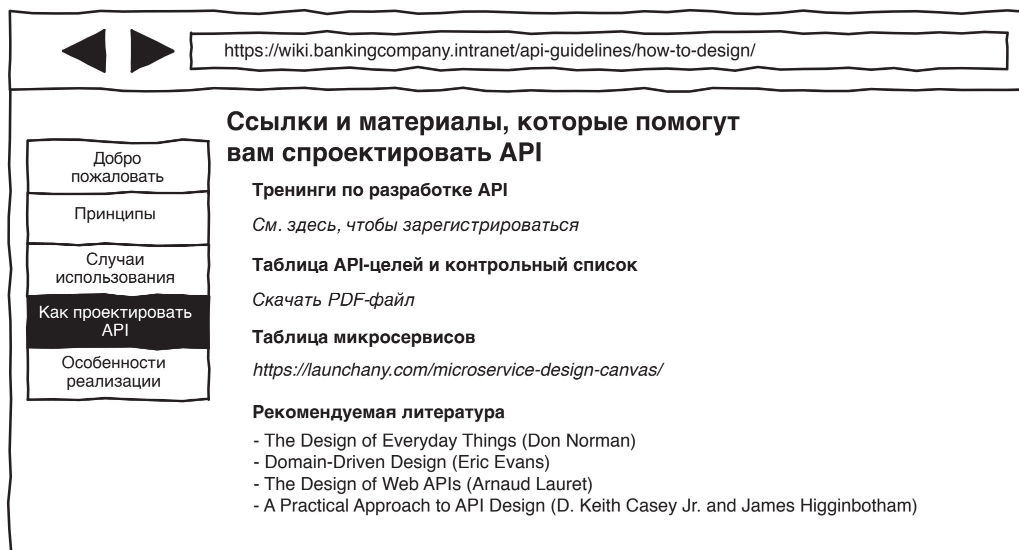



**Рис. 13.4.** Страница Как проектировать API в руководстве по проектированию банковской бомпании

Существует множество соображений относительно создания API, которые можно включить в свое руководство.

### Больше чем руководство по проектированию интерфейса

Расширенное руководство по проектированию API может содержать информацию о безопасности, проблемах сети или реализации, как показано на рис. 13.5. Оно может включать в себя сведения о стандартных данных, прикрепленных к токенам безопасности, которые поток OAuth использует в данном контексте, или как настроить фреймворки для получения ожидаемых результатов при реализации API.

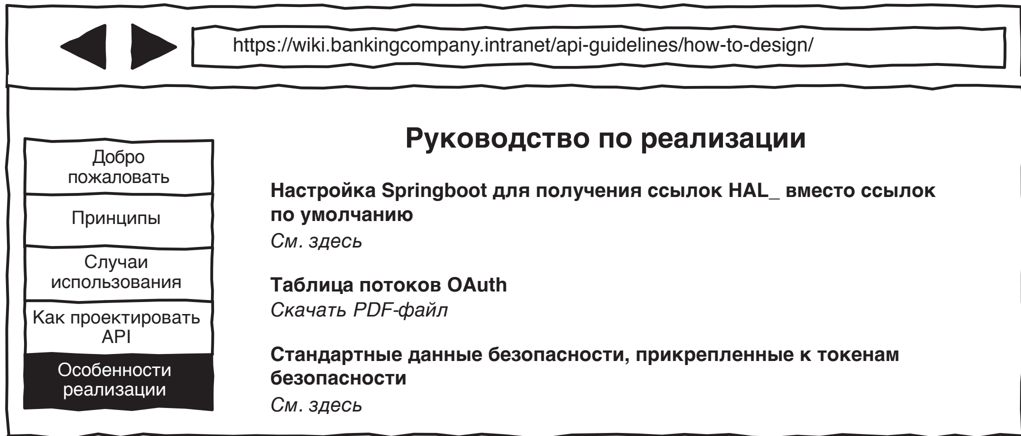



**Рис. 13.5.** Страница Вопросы реализации в руководстве по разработке банковской компании

По сути, вы можете поместить все, что считаете нужным, в свое руководство по проектированию API, чтобы обеспечить согласованность и помочь людям, которые проектируют и создают API. Но до сих пор мы говорили только о том, что включать в руководство, а не о том, как создавать это. Мы займемся этим далее.

## 13.2.2 Постоянное создание руководств

Создание руководства по проектированию API требует от нас не только написания, но и их доработки информировать людей о них, заставляя принять их.

### Начните с малого и точного

При написании руководства по проектированию API не пытайтесь создавать лучшее руководство из возможных, охватывающее все возможности и все пограничные случаи за один раз. Вы будете тратить свое время, и в конечном итоге у вас получится бессмысленная документация низкого качества. Вместо этого начните с освещения базовых, необходимых тем простым способом. Стремитесь к полноте и точности. Сейчас не время начинать изобретать причудливые способы использования HTTP!

### Веб-концепции

Проектирование API и создание руководства по его проектированию требует глубокого понимания веб-концепций, таких как заголовки HTTP и коды состояния. Но не всегда просто обнаружить, какой RFC содержит самый актуальный источник истины относительно конкретного заголовка или другой концепции, чтобы можно было правильно использовать его. К счастью, Эрик Уайлд создал сайт Web Concepts (http://webconcepts.info/), чтобы помочь справиться с этой задачей.

*«Единый интерфейс» сети основан на большом и растущем наборе спецификаций. Эти спецификации устанавливают общие концепции, которые могут использовать поставщики и потребители веб-сервисов. Web Concepts предоставляет обзор этих концепций и определяющих их спецификаций.*

*Одним из примеров того, как это работает, является кеширование в HTTP/1.1, которое определяет 5 полей заголовков HTTP, 7 кодов предупреждений HTTP и 12 директив кеширования. Web Concepts предоставляет структурированный, быстрый и взаимосвязанный обзор этих и многих других концепций, которые вместе образуют «веб-поверхность».*

*Эрик Уайлд*

Как упоминалось в разделе 6.1.4, вас могут вдохновлять и другие, в особенности ваши любимые API. Если есть API, которые вы действительно любите использовать, почему бы не скопировать их стиль? Точно так же вы можете воспользоваться существующими руководствами для API, которые открыто распространяются другими компаниями, вместо того чтобы изобретать свои собственные.

## API Stylebook

API Stylebook (http://apistylebook.com) – это веб-сайт, на котором я собираю и анализирую рекомендации по проектированию API. Он направлен на то, чтобы помочь проектировщикам решать вопросы проектирования API и создавать собственные рекомендации, предоставляя быстрый и простой доступ к рассматриваемым в руководстве темам на сайте. Вместо того чтобы заново изобретать колесо или часами искать в интернете, проектировщики API могут просматривать сайт API Stylebook, чтобы быстро находить решения и черпать вдохновение из существующих рекомендаций.

Если ваше руководство по проектированию API начинается с малого, это означает, что они будут развиваться. Им нужно давать возможность развиваться, чтобы включать туда новое содержимое, связанное с ситуациями, которые не были рассмотрены в начале.

### *Развиваться, адаптировать и исправлять*

Вместо того чтобы постепенно добавлять все, что приходит вам в голову, в своем руководстве по проектированию API, идея состоит в том, чтобы добавлять только проверенный на практике контент, который использовался для решения реального вопроса, связанного с проектированием. Руководство также можно исправлять, если некоторые правила, варианты использования или другое содержимое окажется бременем при применении или неудобным в долгосрочной перспективе. И вы можете понять, что вам нужны разные типы руководств в зависимости от контекста. Например, вы можете использовать gRPC API для внутреннего использования, а REST API – в интернете.

Очевидно, что разработка руководства по проектированию API означает работу с критическими изменениями, версиями и журналами изменений. Руководства в основном описывают API, поэтому при изменении существующих правил помните о том, что вы узнали из раздела 9.1. Вы должны знать о возможных критических изменениях и управлять версиями вашего руководства соответствующим образом. Не забудьте перечислить изменения, внесенные в каждую версию, чтобы разработчики не пытались делать то, что было описано в предыдущей версии.

#### *Коллективная сборка, без догм*

Тот факт, что вы пишете руководство, не означает, что все проектировщики API в вашей команде, организации или компании волшебным образом узнают о них или примут их. Что произойдет, если вы напишете руководство и просто положите его на полку или разместите в «Википедии»? Абсолютно ничего. Вам придется общаться; вам придется продвигать его. Вы должны сообщить проектировщикам, что существует руководство и что оно существует для того, чтобы помочь людям не тратить время на повторное изобретение колеса и помочь им решить проблемы проектирования. Вам придется объяснить, почему важна согласованность и, возможно, почему было выбрано определенное правило (кстати, в своих правилах стоит написать, почему они существуют!).

Но просто сказать все это может быть недостаточно, чтобы заставить людей принять ваши рекомендации. В конце концов, у всех нас есть свои предпосылки и точки зрения. Возможно, у вас возникнет искушение напрямую навязать использование своего руководства и перейти к темной стороне управления. Пожалуйста, не делайте этого.

Создание руководства по проектированию API – это не то, что должен делать отдельный проектировщик или, что еще хуже, человек, который никогда не запускал API в производство. Руководство должно создаваться коллективно настоящими проектировщиками для реальных проектировщиков, без догмы, всегда готовыми корректировать/добавлять/исправлять/развиваться и даже отступать при необходимости. Руководство по проектированию API не должно навязываться «полицией API» любой ценой, и оно не должно отражать идеи и предпочтения отдельного человека. Понадобилась бы целая книга, чтобы полностью объяснить, как именно действовать, но я думаю, вы уловили суть.

## 13.3 Проверка API

Что может пойти не так при создании API? Все. Даже если вы прочитали эту книгу. Ну, если вы прочитали ее, риск может быть не так велик, но все равно что-то может пойти не так, если API не будут проверены. Давайте проиллюстрируем это с помощью еще одной истории.

Однажды в банковской компании кто-то решил, что им нужна возможность отправлять электронные письма клиентам. Поэтому они разработали API для этого. Идея заключалась в том, чтобы позже добавить

другие функции, такие как отправка SMS-сообщений или уведомлений. Поначалу звучало неплохо.

Дизайн API был передан в гильдию проектировщиков API, чтобы посоветоваться. Но, прежде чем даже проанализировать предложенный дизайн, кто-то из гильдии спросил, почему была использована цель отправки электронной почты и в каком контексте. Ответ заключался в том, что, когда клиенты меняли свои личные данные через мобильное приложение или веб-сайт, например изменяя свой почтовый или электронный адрес, им нужно было получить электронное письмо, подтверждающее, что изменения были внесены, и сообщающее о том, что, если они не просили ничего не менять, им нужно немедленно связаться с банковской компанией. Цель должна была вызываться сайтом или мобильным приложением после вызова цели «Обновить клиента». Это считалось проблемой, поскольку могло означать, что потребителям следовало знать, что электронное письмо должно быть отправлено при обновлении информации о клиенте. Анализ дизайна выявил и другие существенные проблемы.

Цель «Отправить электронное сообщение» была представлена в виде запроса `POST /send-email`, ожидающего адрес электронной почты и сообщение. Такой запрос не соответствует руководству по API компании: потребители должны знать адрес электронной почты клиента и какое сообщение должно быть отправлено, чтобы использовать его. Но все это было ничтожным по сравнению с дырой в системе безопасности, которую мог спровоцировать этот API. Он предоставил бы в интернете способ рассылки писем любого содержания кому угодно от имени банковской компании!

API-интерфейсы нужно тщательно проверять (анализировать, проверять, тестировать и т. д.) на различных этапах их жизненного цикла, чтобы гарантировать их работу по назначению. Прежде всего должны быть четко определены потребности (они должны быть хорошо поняты) и необходимо убедиться, что создание API на самом деле является лучшим решением.

Как только API будет спроектирован, необходимо линтировать его интерфейсный контракт, чтобы убедиться, что в нем нет ошибок и в случае модификаций в нем не будет критических изменений. Затем его нужно проверить с точки зрения поставщика: соответствует ли он установленным потребностям и является ли он безопасным, реализуемым и расширяемым? Его также нужно проверить с точки зрения потребителя: понятен ли он и применим ли он в контексте потребителя? И наконец, что не менее важно, после реализации API необходимо протестировать его реализацию, чтобы убедиться, что он действительно предоставляет ожидаемый интерфейсный контракт.

Проектировщики API обычно принимают активное участие или, по крайней мере, имеют право голоса во всех этих проверках. Очевидно, что разработчики должны участвовать в проверке API, над которыми они работают, но они также могут помочь с проверкой других API. Получение отзывов от других всегда полезно и помогает гарантировать ка-

чество. Кроме того, получение отзывов от опытных проектировщиков API помогает новичкам совершенствоваться. И наоборот, помощь другим проектировщикамI приносит пользу, потому что это хороший способ расширить кругозор и открыть новые варианты и идеи для использования дизайна.

Будучи проектировщиком API, ваша задача – задавать вопросы, проверять, исследовать, оценивать и анализировать все, что касается разработки ваших API. Но вы также должны следить за общим контекстом – всем, что связано с проектированием, – чтобы убедиться, что выбранные направления верны. (Помните главу 11, где вы узнали, как разрабатывать API в контексте?)

Идея состоит в том, чтобы не выступать в роли враждебного эксперта, который все знает, и навязать свою волю всем остальным, а скорее участвовать в развитии API вне их проектирования наиболее подходящим образом, как это стремятся делать все, кто работает в команде или организации.

### 13.3.1 Оспаривание потребностей и их анализ

В случае использования цели для отправки электронной почты вопрос, почему и в каком контексте она должна использоваться, показал, что воображаемая потребность не была реальной и что предлагаемое решение вообще не было обеспечено. Реальная необходимость заключалась не в том, чтобы отправить электронное письмо, а в том, чтобы уведомлять клиентов об активности в их профилях. И в итоге реализованным решением стало не REST API, а система публикации и подписки (см. раздел 11.3.2).

В конечном итоге было решено создать новое серверное приложение для уведомления клиентов, основываясь на событиях, отправляемых приложением клиента. С помощью этого решения клиентское приложение, которое предоставляет доступ к API при использовании цели «Обновить клиента», отправляет событие, уведомляющее об обновлении клиента, при изменении клиентских данных. Приложение для уведомления клиентов подписывается на эти события. Оно содержит все знания о том, как уведомить клиентов, например канал, который они предпочитают (электронная почта, уведомление через мобильное приложение или SMS) и связанные данные, такие как их адрес электронной почты и номер телефона. Оно реагирует только на выбранные события: в первой версии это – обновление данных клиента.

Такая несвязная архитектура разделяет проблемы, является гибкой и может легко расширяться, в то же время будучи безопасной и удовлетворяя фактические потребности. Реализованное решение полностью отличается от того, что представлялось на первый взгляд. Хорошо, что гильдия разработчиков API банковской компании провела проверку, перед тем как начать писать код! API для отправки электронной почты был разработан впустую, и этой траты времени можно было бы избежать, если бы проверка была проведена раньше.

В главе 2 вы узнали, что API должен удовлетворять потребности потребителей, чтобы избежать создания ужасных API-интерфейсов по типу Кухонного радара или, что еще хуже, небезопасных целей для отправки электронной почты, как мы только что видели. Такие предполагаемые потребности потребителей нужно тщательно оценивать и оспаривать, чтобы определить реальные потребности и найти адекватные, реализуемые и безопасные решения.

Поэтому, прежде чем думать о том, какие URL-адреса или методы HTTP использовать, прежде чем даже думать о заполнении таблицы API-целей, вам необходимо оспорить и проанализировать выявленные потребности. И сделать это нужно как можно раньше. Как только потребности будут идентифицированы, их нужно обсудить и оценить, до того делать что-либо еще. На рис. 13.6 показан пример контрольного списка, который можно использовать для оспаривания и анализа выявленных вами потребностей; он основан на содержимом этой книги, но вы можете спокойно адаптировать его к своей ситуации.

Анализ потребностей не зарезервирован для API. Четкое определение потребностей или проблем, которые необходимо решить, должно осуществляться при создании чего-либо. Не существует единого наиболее подходящего подхода для оспаривания и анализа потребностей; вы можете использовать свой любимый метод.

Вопросы типа «почему что-то должно быть сделано?», «каков контекст?» и «как это будет использоваться?» обычно помогают определить реальную потребность или потребности, скрытые первым требованием. Метод «Пять почему» также является хорошим способом сделать это: вы просто спрашиваете «почему?» и получаете ответ, потом снова спрашиваете «почему?», чтобы углубиться в анализ. Сделать это пять раз подряд обычно достаточно, чтобы найти истинную корневую потребность. Все эти элементы влияют на разработку решения и API.

После этого вы также должны изучить контексты потребителя и поставщика (см. главы 10 и 11) и подумать о безопасности (см. раздел 8.1.4). Все это итерационный процесс, который должен проводиться с различными участниками (теми, кто вовлечен в определение потребностей, реализацию, вопросы безопасности и т. д.).

Четкое определение потребностей и изучение контекста обеспечит разработку наиболее подходящего решения, будь то добавление новых целей в существующий API, создание нового API или создание чего-то, что не является API или где API является лишь частью решения. Если API *и есть* решение, наличие четко определенных потребностей гарантирует, что цели, которые API помогает достичь, на самом деле верные.

**Контрольный список потребностей и их анализ**
(из книги «Проектирование веб-API»)

**Анализ потребностей**

| Вопрос / Инструмент | Описание |
|---|---|
| Что вы хотите делать? Каков контекст? | Это первоочередные вопросы, которые нужно задать, чтобы получить представление о потребностях и начать понимать их контекст |
| Как «это» будет использоваться? | Не забудьте явно спросить, как предполагается использовать это «решение»; это дает более подробную информацию о контексте |
| Кто является целевым потребителем? | Знание того, кто является потребителями и конечными пользователями, даст направление относительно проектирование (например, с использованием определенного отраслевого стандарта) и может поднять вопросы, касающиеся о безопасности (см. раздел, посвященный безопасности, ниже) |
| Кто является целевым конечным пользователем? | Знание того, кто является возможными конечными пользователями (если таковые имеются), может дать направление относительно проектирования (например, поддержка интернационализации и локализации) и безопасности (см. раздел, посвященный безопасности, ниже) |
| Пять почему | Пять почему – простой инструмент, который можно использовать для определения первопричины «проблемы», потребности (потребностей) в нашем случае и, следовательно, выявления реальной потребности (потребностей). Спросите «почему?» и получите ответ, затем снова спросите «почему?», чтобы углубиться в анализ. Сделайте это пять раз подряд. Обычно этого достаточно, чтобы найти истинную потребность |
| Таблица API-целей | Используйте таблицу целей API, как только у вас появится подходящий обзор «реальных» потребностей |

**Изучение контекста (контекстов) потребителя (потребителей)**

| Вопрос | Описание |
|---|---|
| Кто является целевым потребителем? | Знание того, кто является потребителями, даст направление относительно проектирования (например, используя отраслевой стандарт) и может поднять вопросы, касающиеся о безопасности (см. раздел, касающийся безопасности ниже) |
| Кто является целевым конечным пользователем? | Знание того, кто является возможными конечными пользователями (если таковые имеются), может дать направление относительно проектирования (например, поддержка интернационализации и локализации) и безопасности (см. раздел, посвященный безопасности, ниже) |
| Есть ли у потребителей ограничения? | Например, некоторые потребители могут использовать только формат XML или не могут использовать HTTP- метод PATCH. |
| Существует ли отраслевой стандарт? Используется ли он целевыми потребителями? | Вместо того, чтобы изобретать колесо, которое никто не захочет использовать, отдавайте предпочтение стандартам, если они существуют и фактически используются целевыми потребителями. Обратите внимание, что API может поддерживать разные форматы (стандартный и пользовательский), поэтому вы также можете предлагать разные API разным типам клиентов |
| Какова их сетевая среда? | В зависимости от сетевой среды при проектировании API могут потребоваться дополнительные меры предосторожности. Также могут потребоваться Experience API (или BFF-компоненты ) |

**Изучение контекста поставщика**

| Вопрос | Описание |
|---|---|
| Какие существующие системы / API / команды / партнеры необходимы «под капотом»? | Раннее выявление зависимостей позволяет избежать ситуации, когда вы слишком поздно обнаруживаете, что они не могут делать то, что должны были сделать, а также их ограничений. Зависимости могут быть внутренними системами, командами с работающими в них людьми, или системами партнеров. Кроме того, новые потребности можно удовлетворить, используя существующие API, как они есть, или вам может потребоваться расширить их. |
| Существуют ли технические ограничения? | Зависимости могут иметь ограниченные возможности или особенности, которые будут влиять на дизайн (не запускаться в режиме 24/7, только асинхронность, длительное время обработки, длительное время отклика, отсутствие масштабируемости …) |
| Существуют ли функциональные ограничения? | Зависимости могут иметь существующие бизнес-правила, которые несовместимы с потребностями |
| Участвуют ли в этом люди? | Если внутри находится человеческий процесс, возможно, его нужно автоматизировать или дизайн нужно будет адаптировать |
| Какой тип (типы) передачи данных необходим? | В зависимости от потребностей и контекста, вы должны определить тип (типы) передачи данных, который будет необходим (синхронный, асинхронный, потоки, события) |
| Какой тип (типы) API необходим? | В зависимости от потребностей и контекста, вы должны определить тип (типы) API, который вам понадобится |
| Является ли API решением? | В зависимости от потребностей и контекста, API может не являться решением или может быть только частью решения |

**Изучение проблем безопасности**

| Вопрос | Описание |
|---|---|
| Имеет ли API дело с конфиденциальными вещами? | В зависимости от конфиденциальности данных и действий, а также от того, кто является потребителями и конечными пользователями, может потребоваться адаптировать дизайн (а также реализацию) |

**Рис. 13.6. Контрольный список потребностей, которые можно оспорить**

### 13.3.2 Линтирование

Был ли допустимым предложенный запрос `POST /send-email`, ожидающий необязательного свойства `email` в виде строки и msg в качестве числа? Он использует путь к ресурсу, который не похож на тот, который вы учились использовать в разделе 3.2.3, но, возможно, в руководстве по проектированию API банковской компании говорится, что API должны быть основаны на функциях (см. раздел 11.3). Тем не менее, чтобы было ясно, может быть, стоит назвать `msg message` и оно должно быть строкой? И разве это нормально, чтобы все ожидаемые свойства были необязательными? Линтирование предложенного варианта даст ответы на все эти вопросы:

> *«Lint – первоначально – статический анализатор для языка программирования C, который сообщал о подозрительных или непереносимых на другие платформы выражениях».*
>
> *«Википедия»*

Как и код, дизайн API может содержать ошибки (например, неправильный тип свойства) и, возможно, его нужно писать (проектировать) в соответствии с такими соглашениями, как использование имени свойства `errorMessage` вместо `err_msg`. Линтирование API поможет обнаружить такого рода ошибки.

Линтирование включает в себя проверку на предмет наличия ошибок в проектировании, а также проверку с целью убедиться, что API соответствует руководству по проектированию и что он согласован со всеми ранее существовавшими элементами (скоро мы рассмотрим, что это такое). При линтировании вы также должны проверить его безопасность и документацию. То есть, по сути, вы линтируете все, что будет помещено в файл описания API; нужно проверить каждый аспект контракта интерфейса. На рис. 13.7 показан пример контрольного списка.

По сути, вы должны проанализировать каждую модель и ее свойства, а также каждую цель, ее параметры и ответы. Вы также должны проанализировать потоки целей и определения безопасности. Большинство из этих проверок довольно просты и двоичны, но есть три типа проверки, где нужно быть особенно внимательными: проверка документации, проверка потоков и (что наиболее важно) проверка согласованности с существующими ранее элементами.

Проверка документации по контракту интерфейса важна, потому что наличие исчерпывающе документированного контракта облегчит рассмотрение проекта как с точки зрения поставщика, так и с точки зрения потребителя. Мы поговорим об этом в последующих разделах (13.3.3 и 13.3.4).

Проверка потоков – это то, что вы научились делать в разделах 2.3 и 3.3.4. Это очень просто, но важно убедиться, что API действительно работает. Было бы неловко (если не сказать катастрофично), если бы потребители не могли использовать API, потому что им не удалось предоставить какие-то параметры.

Проверка согласованности с существующими ранее элементами – возможно, самая сложная проверка, но она очень важна. Проверка на предмет того, что имя, модель данных или поведение цели соответствуют ранее существовавшим элементам, требует концентрации и подробных знаний о том, что на самом деле находится внутри и за пределами API, который вы анализируете. В противном случае в конечном итоге у вас может получиться, например, три разных формата адресов, два разных стандартных способа для обозначения стран, слишком много способов, для того чтобы показать, что сумма является недопустимой, и полностью индивидуальный способ обозначения телефонных номеров.

| Цели | |
|---|---|
| **Проверка** | **Описание** |
| Упорядоченно по категориям | Это не обязательно, но классификация целей облегчает понимание API |
| Допустимый путь | Формат пути соответствует руководству и согласуется с уже существующими элементами. Он также соответствует типу сообщения об успешном выполнении. Если параметры пути определены, они должны присутствовать |
| Допустимый HTTP- метод | HTTP-метод соответствует руководству и уже существующим элементам и соответствует тому, что должна делать цель |
| Обработка успешного результата и ошибок | Цель возвращает все необходимые ответы для обработки как успешного результата, так и ошибок |
| Защищенность | Каждая цель покрыта механизмом безопасности |

| Параметры цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимое имя | Имя параметра соответствует руководству и согласуется с уже существующими элементами |
| Допустимый требуемый статус | Статус параметра «необязательный /обязательный» соответствует руководству и согласуется с уже существующими элементами и его расположением (параметр пути должен быть обязательным). Если все свойства параметра тела являются необязательными при создании / замене, то, вероятно, это ошибка (но такое может произойти) |
| Допустимый тип и формат | Тип и формат параметра соответствуют руководству и уже существующим элементам и являются допустимыми в соответствии с его местоположением (например: в параметрах запроса нет объектов, нет запрещенных символов URL-адресов в параметрах пути) |
| Допустимое описание | Удобные для восприятия человеком (свойство description) и машиночитаемые (минимальное количество и т. д.) описания параметра являются допустимыми и точными, они соответствуют руководству и уже существующим элементам. |
| Допустимое расположение | Расположение параметра соответствует руководству, согласуется с уже существующими элементами и соответствует методу состояния HTTP (никакого параметра тела для запроса на удаление, например) |
| Возможность предостав-ления | Потребители должны быть в состоянии предоставить параметры каждой цели (потому что они их знают или получили их от другой цели) |

| Ответы цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимое состояние HTTP | Код состояния ответа соответствует руководству и уже существующим элементам и типу ответа (успех, сбой у потребителя, сбой у поставщика). |
| Допустимое тело | Тип и формат тела ответа соответствуют руководству и уже существующим элементам и соответствуют состоянию HTTP (например, стандартная модель ошибки для кода 4XX или 5XX). |
| Допустимые заголовки | Имена, типы и форматы заголовков ответа соответствуют руководству и уже существующим элементам. |
| Использование | Данные, предоставляемые в ответе каждой цели, на самом деле используются |
| Допустимое описание об успешном результате | Удобные для восприятия человеком (свойство description) и машиночитаемые (определение модели) описания ответа, его тела и заголовков являются допустимыми и точными, соответствуют руководству и уже существующим элементам |
| Допустимое описание об ошибке | Удобные для восприятия человеком (свойство description) и машиночитаемые (определение модели) описания ответа, его тела и заголовков являются допустимыми и точными, соответствуют руководству и уже существующим элементам. Подробная информация о возможных ошибках должна предоставляться в удобных для восприятия человеком и, если возможно, машиночитаемых описаниях |

**Контрольный список по линтингу**
(из книги «Проектирование веб-API»)

| Изменения (относится ко всем темам) | |
|---|---|
| **Проверка** | **Описание** |
| Внесены критические изменения | Если модификация предполагает критическое изменение, проверьте, действительно ли это необходимо. Если да, примените свою политику управления версиями. |

| Определения безопасности API | |
|---|---|
| **Проверка** | **Описание** |
| Использование | Каждое определение безопасности должно быть использовано. Если нет, проверьте, не осталось ли оно в цели. Если нет, удалите его |
| Допустимость | Соответствующая машиночитаемая информация должна предоставляться в соответствии с типом безопасности и соответствовать руководству |
| Допустимое описание | Должно быть предоставлено релевантное описание каждой группы. |

| Потоки целей API | |
|---|---|
| **Проверка** | **Описание** |
| Допустимость | Поток целей (вариант использования) и его поведение соответствует руководству и уже существующим элементам |
| Допустимое описание | Поток целей (вариант использования) точно описан в API или описании целей |

| Модели | |
|---|---|
| **Проверка** | **Описание** |
| Использование | Все определения модели должны быть использованы. Если определение не используется, возможно, оно было оставлено в цели. Если нет, удалите его |
| Допустимое имя | Имя модели соответствует руководству и согласуется с уже существующими элементами |
| Допустимая структура данных | Структура /организация данных и глубина модели соответствуют руководству и согласуются с уже существующими элементами. |
| Допустимое описание | Описание модели, понятное человеку (свойство description) присутствует (и является точным) в случае необходимости |

| Свойства модели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимое имя | Имя свойства соответствует руководству и согласуется с уже существующими элементами |
| Допустимый требуемый статус | Статус свойства «необязательное/обязательное» соответствует руководству и согласуется с уже существующими элементами. Если все свойства являются необязательными, вероятно, это ошибка (но такое случается) |
| Допустимый тип и формат | Тип данных (атомарный или модель) и формат свойства (например, данные в формате ISO 8601) являются допустимыми в соответствии с его именем, значением или описанием. Они также должны соответствовать руководству с его именем и быть согласованными с уже существующими элементами |
| Допустимое описание | Удобные для восприятия человеком (свойство description) и машиночитаемые (минимальное количество и т. д.) описания параметра являются допустимыми и точными, они соответствуют руководству и согласуются с уже существующими элементами |

**Рис. 13.7.** Линтирование API – пример контрольного списка

Предварительно существующие элементы в основном происходят из самого API, других API организации и стандартов; это могут быть имена, типы, модели данных и даже поведение. Чтобы убедиться, что API согласован с ранее существовавшими элементами в самом API или других API из той же организации, можно частично полагаться на чутье разработчика API. Кроме того, если вам посчастливилось участвовать в проверке других API, вы, возможно, помните, что вы видели. Но трудно идти в ногу со всем, что может происходить на поверхности API для организации, поэтому вам также следует использовать доступную документацию по API. Чтобы облегчить проведение такого анализа, может быть полезным задействовать механизм поиска по документации всех ваших API.

Что касается идентификации стандартов, которые можно взять вместо пользовательских форматов, все зависит от вас, от вашего проектировщика API и вашей любимой поисковой системы. Существуют очевидные стандарты, о которых вы слышали, например ISO 8601 для описания форматов дат и ISO 4217, устанавливающий коды валют, но вы не можете знать *все* существующие стандарты. Например, знаете ли вы, что существует стандарт для представления телефонных номеров? Это формат E.164, определенный Международным союзом электросвязи (МСЭ).

Всегда нужно проверять, существует ли стандарт для представления данного элемента данных. Если вы найдете его, сразу же добавьте его в свои правила. Поначалу это может быть обременительно; но в конечном итоге это облегчит вашу жизнь, поскольку вы быстро внесете в каталог все стандарты, которые необходимы вам для ваших API. И в случае изменения существующего API не забудьте проверить, вносите ли вы какие-либо критические изменения. Если их нельзя избежать, вам придется применить свою политику управления версиями.

Линтирование можно частично автоматизировать, запустив некоторые инструменты в машиночитаемом формате описания API, но в этом все равно должен участвовать человек. Достаточно просто проверить форматы пути или авторизован ли код состояния HTTP, но довольно сложно проверить, насколько релевантно удобное для восприятия человеком описание. Несмотря на то что полностью автоматизировать процесс невозможно, настоятельно рекомендуется воспользоваться преимуществами автоматизации, чтобы проверяющие могли сосредоточиться на проверках, где действительно нужны люди.

Наконец, имейте в виду, что при линтировании проверяется только форма, а не сущность. Это лишь гарантирует, что спроектированный контракт интерфейса соблюдает определенные правила проектирования и абсолютно не подтверждает, что API отвечает всем требованиям поставщика и что потребители действительно захотят и смогут его использовать. После этого API нужно тщательно проверить как с точки зрения поставщика, так и с точки зрения потребителя.

### 13.3.3 Проверка дизайна с точки зрения поставщика

Тот факт, что вы определили реальные потребности и контекстные элементы, необходимые для проектирования API, и его дизайн полностью

проанализирован, не означает, что все в порядке с точки зрения поставщика. Вы должны уточнить у всей команды, что получившийся дизайн API действительно отвечает всем требованиям поставщика. Возможно, вы хотите подтвердить, что проект удовлетворяет всем выявленным потребностям и является безопасным, реализуемым и расширяемым. На рис. 13.8 показан еще один пример контрольного списка на базе содержимого этой книги, который можно использовать для этой цели. Опять же, не стесняйтесь адаптировать его к своим потребностям.

**Проверка с точки зрения поставщика**
(из книги «Проектирование веб-API»)

| Цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимая цель | Соответствуют потребностям |
| Безопасный путь | Не предоставляют доступ к ненужным конфиденциальным данным, а предоставляют доступ к конфиденциальным данным с защищенными представлениями. |
| Исчерпывающие сообщения об успешном результате | Соответствуют потребностям (определены все ожидаемые ответы об успешном результате) |
| Исчерпывающие сообщения об ошибке | Соответствуют потребностям (обрабатывают все возможные ошибки управления поверхностью, функциональные ошибки, ошибки безопасности и технические ошибки) |
| Защищенность | Охвачены ожидаемой группой (группами) и механизмом (механизмами) безопасности, и не предоставляют неправомерного доступа |
| Возможность реализации | Могут быть реализованы (напрямую или через зависимость) |
| Допустимые оценки производительности | Соответствуют потребностям (следовательно «могут быть реализованы»). |
| Расширяемость | Разумно принимают во внимание будущие изменения |

| Параметры цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимый требуемый статус | Соответствуют потребностям |
| Допустимый тип и формат | Соответствуют потребностям. |
| Безопасные данные | Не требуют ненужных конфиденциальных данных, а требуют конфиденциальных данных с защищенными представлениями или местоположениями |
| Допустимое местоположение | Соответствуют потребностям и вопросам безопасности (см. Безопасные данные). URL-адрес не может содержать конфиденциальные данные |
| Расширяемость | Разумно принимают во внимание будущие изменения |

| Ответы цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимые данные об успехе | Соответствуют потребностям |
| Безопасные данные об успехе | Не предоставляют доступ к ненужным конфиденциальным данным, а предоставляют доступ к конфиденциальным данным с защищенными представлениями |
| Существующие данные об успехе | Фактически могут быть предоставлены реализацией (напрямую или через зависимость) |
| Допустимые данные об ошибках | Соответствуют потребностям и являются исчерпывающими |
| Безопасные данные об ошибках | Не предоставляют доступ к ненужным конфиденциальным данным, а предоставляют доступ к конфиденциальным данным с защищенными представлениями |
| Существующие данные об ошибке | Фактически могут быть предоставлены реализацией (напрямую или через зависимость) |
| Расширяемость | Разумно принимают во внимание будущие изменения |

| Определения безопасности API | |
|---|---|
| **Проверка** | **Описание** |
| Релевантные группы | Определенные группы соответствуют требованиям разделения безопасности |

| Потоки целей API | |
|---|---|
| **Проверка** | **Описание** |
| Допустимое назначение | Соответствуют потребностям |
| Допустимые оценки производительности | Соответствие потребностям (накопленная производительность) |
| Расширяемость | Разумно принимают во внимание будущие изменения |

| Модели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимая структура данных | Соответствует потребностям (на самом деле представляет ожидаемую концепцию) |
| Расширяемость | Разумно принимают во внимание будущие изменения |

| Модели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимый требуемый статус | Соответствуют потребностям |
| Допустимый тип и формат | Соответствуют потребностям |
| Допустимое значение | Соответствуют потребностям |
| Безопасное значение | Не предоставляют доступ к ненужным конфиденциальным данным, а предоставляют доступ к конфиденциальным данным с защищенными представлениями |
| Существующее значение | Фактически может быть предоставлено реализацией (напрямую или через зависимость) или запрошено, если используется в качестве параметра |
| Расширяемость | Разумно принимают во внимание будущие изменения |

**Рис. 13.8. Проверка дизайна с точки зрения поставщика**

Если вы использовали таблицу API-целей, создали исчерпывающую документацию и применили то, что узнали в этой книге, для проектирования контракта интерфейса, такая проверка вообще не должна быть проблемой. Таблица целей API поможет связать начальные потребности и цели. Здесь важна документация, чтобы всем вовлеченных лицам, которые не имеют глубоких познаний об API, было проще проводить проверку.

Различные потоки целей должны соответствовать выявленным потребностям, а каждая цель должна вести себя так, как и ожидалось, возвращая правильные данные и инициируя правильные ошибки. Каждый элемент должен быть достаточно расширяемым (вспомните раздел 9.3). Кроме того, не забудьте проверить, что каждая цель, поведение или возвращаемый ответ на самом деле могут быть обработаны реализацией и что производительность каждой цели и потока соответствует ожиданиям. Но эта проверка касается только точки зрения поставщика; также не нужно забывать о том, что нужно проверить, все ли в порядке с точки зрения потребителя.

### 13.3.4 Проверка дизайна с точки зрения потребителя

Последнее, но не менее важное в анализе дизайна, – это проверка с точки зрения потребителя. Является ли этот API простым в использовании, понятным и эффективным? Следует ли позаботиться о том, чтобы не раскрывать чрезмерно точку зрения поставщика? Если нет, то все усилия, которые вы приложили к созданию API, ничего не будут стоить, потому что никто не захочет использовать полученный продукт (см. раздел 1.2.3). Надеюсь, все, что вы узнали из этой книги, сделает эту проверку проще некуда. На рис. 13.9 показан пример контрольного списка для проверки дизайна API с точки зрения потребителя на основе содержимого этой книги. Как обычно, не стесняйтесь адаптировать его под свои нужды.

Во время этой проверки вы должны поставить себя на место потребителя, который ничего не знает об API. Убедитесь, что каждая цель, ее параметры и ответы (особенно ошибки) имеют смысл для потребителей и не являются неприятным изложением того, что происходит «под капотом» (точка зрения презираемого провайдера). Убедитесь, что каждый обязательный параметр действительно необходим для запроса минимальных данных. Обратите особое внимание на имена и описания: убедитесь, что это не жаргон поставщика. Проверьте также, что целевые потоки просты и эффективны, что здесь не слишком много шагов и есть цели, которые предотвращают ошибки. И не забудьте проверить, что оценки производительности (для отдельных целей и потоков целей) допустимы как для базовых, так и для комплексных вариантов использования (см. раздел 10.1).

**СОВЕТ.** Не стесняйтесь представить свой дизайн людям за пределами команды и, если это возможно, потенциальным потребителям на рассмотрение; их отзывы будут полезны и повысят качество работы.

Наконец, после того как все проверки будут выполнены – линтирование (см. раздел 13.3.2), проверка с точки зрения поставщика и с точ-

ки зрения потребителя, – API наконец может быть реализован. Но это не значит, что ваша работа должна на этом завершиться. Возможно, вам придется убедиться, что разработанный интерфейсный контракт и в самом деле работает, как и ожидалось.

### 13.3.5 Проверка реализации

Я не буду объяснять, как на самом деле протестировать реализацию API, потому что это выходит за рамки этой книги. Для этого существует множество инструментов, и некоторые из них даже позволяют создавать тесты на основе файлов описания API. Но этих тестов обычно недостаточно для проверки того, что реализация выполняет все, что ожидалось, поэтому придется писать их.

**Проверка с точки зрения потребителя**

| Цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимый путь | Структуру пути и используемые имена или параметры можно легко понять |
| Допустимое назначение | Значимы для потребителей |
| Допустимые оценки производительности | Соответствуют потребностям потребителя |

| Параметры цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимый требуемый статус | Минимальные обязательные свойства |
| Допустимое имя | Имя можно легко понять (например, «source» вместо «ts») |
| Допустимый тип и формат | Тип и формат можно легко понять и использовать (например, даты в формате ISO 8601 вместо временных отметок Unix, «CHECKING» вместо «2») |
| Допустимое значение | Значения готовы к использованию (номера счетов вместо технических идентификаторов, например) |
| Возможность предоставления | Потребители должны быть в состоянии предоставить параметры каждой цели (потому что они их знают или получили их от другой цели) |

| Ответы цели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимые данные об успехе | Все необходимые данные возвращаются вместе с информативными данными (например, что было сделано и что делать дальше) |
| Допустимые данные об ошибках | Исчерпывающие и информативные ответные сообщения на самом деле помогают решить все проблемы |

| Определения безопасности API | |
|---|---|
| **Проверка** | **Описание** |
| Релевантные группы | Значимы для потребителей |

| Потоки целей API | |
|---|---|
| **Проверка** | **Описание** |
| Предотвращение ошибок | Цели предотвращают ошибки (например, цель, предоставляющая возможные счета назначения для исходного счета при переводе денег) |
| Самое короткое число этапов | Количество этапов является максимально коротким |
| Допустимые оценки производительности | Соответствуют потребностям потребителей |
| Допустимое назначение | Каждый шаг (цель) имеет значение для потребителей |

| Модели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимая структура данных | Адекватная организация, глубина и детализация, способствующая пониманию и использованию. Минимальные свойства для параметров |
| Допустимое имя | Каждый шаг (цель) имеет значение для потребителей |

| Свойства модели | |
|---|---|
| **Проверка** | **Описание** |
| Допустимый требуемый статус | Минимальные обязательные свойства (для параметров) |
| Допустимое имя | Имя можно легко понять (например, «source» вместо «ts») |
| Допустимый тип и формат | Тип и формат можно легко понять и использовать (например, даты в формате ISO 8601 вместо временных отметок Unix, «CHECKING» вместо «2») |
| Допустимое значение | Значения готовы к использованию (например: добавление возраста наряду с датой рождения, кода с локализованными метками, номеров счетов вместо технических идентификаторов) |

**Рис. 13.9.** Проверка дизайна с точки зрения потребителя

Разработчики, отвечающие за реализацию, могут использовать различные уровни тестирования (обычно это модульные тесты и тесты API) для полной проверки API. Здесь я хочу показать вам несколько вещей,

о которых следует особенно позаботиться, и предупредить вас о ловушках.

### *Никогда не обходите тестирование безопасности*

Тестирование безопасности для API является обязательным. Вы должны убедиться, что средства управления доступом и конфиденциальные данные обрабатываются надлежащим образом.

В случае с управлением доступом первый уровень тестирования состоит в том, чтобы гарантировать, что только зарегистрированные потребители могут получить доступ к API и что они не могут ничего делать вне предоставленных групп. Второй уровень тестирования касается контроля прав доступа потребителя и конечного пользователя. Например, если потребитель запрашивает список счетов от имени конечного пользователя, должны быть возвращены только счета этого пользователя. И если тот же потребитель запрашивает счет 12345, API должен вернуть его только в том случае, если у конечного пользователя действительно есть право получить его.

Что касается конфиденциальных данных, вы должны убедиться, что ненужные конфиденциальные данные не возвращаются. Если конфиденциальные данные должны быть запрошены или возвращены, необходимо убедиться, что они надежно защищены (например, с использованием не-конфиденциального представления или шифрования), о чем шла речь в разделе 8.4.

### *Будьте осторожны при использовании сгенерированной документации для проверки реализации*

Некоторые фреймворки реализации позволяют генерировать файл описания API из кода во время выполнения или при сборке приложения. Этот сгенерированный файл можно сравнить с исходным файлом описания API, чтобы проверить, что контракт интерфейса, представленный реализацией, соответствует ожидаемому. Но такую проверку можно сделано только для элементов, которые не происходят из аннотаций, специально созданных для генерирования файла описания.

Если, например, я добавлю в свой код аннотацию, специфичную для описания API, чтобы указать, что возможные значения атрибута type – это checking и savings, нет гарантии, что мой код будет только возвращать эти значения. Эта информация носит только декларативный характер.

Однако некоторые инструменты могут использовать стандартные аннотации, которые являются предписывающими. Например, если я разрабатываю веб-приложение, предоставляющее доступ к API с помощью фреймворка Java Spring Boot, я могу использовать стандартную аннотацию, чтобы объявить, что метод сопоставим с запросом POST /transfers. Такие значения можно использовать при сравнении исходного файла описания API и сгенерированного.

### *Проверьте контракт интерфейса во время выполнения*

Даже если вы найдете магический трюк, фреймворк или инструмент, который позволяет вам генерировать фактически реализованный интерфейсный контракт, вы должны протестировать реализацию во время выполнения, чтобы проверить все ожидаемые варианты поведения. Просто потому, что в сгенерированном и допустимом контракте интерфейса указано, что в случае отсутствия обязательного атрибута возвращается ответ `400 Bad Request`, не означает, что это действительно так.

### *Проверьте характеристики свойств в ответах*

Когда в контракте интерфейса говорится, что свойство является обязательным в возвращаемой модели данных, оно всегда должно возвращаться. То же самое касается ситуации, если описание API предоставляет информацию о минимальных и максимальных значениях, количестве элементов в массивах и т. д. Не забудьте эти тесты.

### *Проверьте всю цепочку сети*

При тестировании своего API вы должны выполнять тестовые вызовы, охватывающие всю цепочку сети перед реализацией. Эта цепочка может включать в себя, например, брандмауэры, прокси-серверы и шлюзы VIP или API. Брандмауэры – хорошие поставщики ошибок; например, неправильно настроенные брандмауэры могут блокировать HTTP-запросы `DELETE` или заменять коды состояния HTTP 5XX на ответы с кодом `500` или возвращать HTML-страницу (без шуток) вместо данных в случае ошибки.

Кроме того, неправильно настроенные шлюзы API могут реализовывать некоторые элементы управления от имени реализации на основе файла описания API, используемого для предоставления доступа к API, но не обязательно самым подходящим образом. Например, они могут возвращать по одной ошибке за раз или использовать неправильный формат ошибки (используя стандартный формат ошибок шлюза, который отличается от формата API или, что еще хуже, HTML-страницу – опять же, без шуток).

## 13.4 Общайтесь и делитесь информацией

Вот мы и подошли к концу; это последний раздел последней главы книги. Я хочу воспользоваться этой возможностью, чтобы кратко рассказать об общении и обмене информацией. Как вы, возможно, заметили, читая последнюю главу, проектировщики API не работают в одиночку. Разработчик API должен работать, по крайней мере, с людьми, которые хотят создавать API или думают, что API может быть решением, потребителями, людьми, отвечающими за разработку реализации, ответственными за безопасность, документирование, и другими проектировщиками API.

Будучи проектировщиком API, вы должны иметь возможность поделиться тем, что вы делаете, своими наработками. Чтобы без труда сделать это, вы должны, по крайней мере, использовать стандартный формат

описания API. Возможно, вы также захотите воспользоваться системой управления исходным кодом, такой как Git, для хранения ваших файлов, использовать вики для менее структурированных описаний, создать собственный каталог API или применить существующий портал для разработчиков.

Для обеспечения согласованности руководства вашей компании (в которые вы вносите свой вклад) должно быть доступны и известны всем проектировщикам API. Все существующие API и модели данных должны быть, по крайней мере, легко доступны (отсюда и система контроля версий, вики или каталог API) и, если это возможно, сделать так, чтобы их можно было найти.

Чтобы работать уверенно, пусть ваши проекты просмотрят коллеги. Это также может быть полезно и информативно для создания сообщества проектировщиков API или участия в нем (или в гильдии API). И, самое главное, убедитесь, что ваши проекты проверены реальными потребителями.

## *Резюме*

- Документация жизненно важна для проектирования, создания и проверки API.
- Согласованность невозможна без руководства по проектированию API и документации.
- Проектирование API не должна выполняться в одиночку: привлекайте других лиц для работы над проверками и/или создавайте сообщество.
- Проектировщики API участвуют во всем жизненном цикле API.
- Чтобы спроектировать эффективные и полезные API-интерфейсы, проектировщики должны оспаривать и подробно анализировать потребности, которые их API призваны удовлетворить.

# Указатель

## 2

200 OK, 76, 79, 80, 81, 123, 134, 164, 188, 197, 210, 218, 283, 323, 324, 355, 360, 362, 364, 366, 368, 375, 405
201 Created, 171, 188, 218, 283, 284, 348, 391
202 Accepted, 171, 188, 218, 284, 348, 350, 391

## 3

301 Moved Permanently, 285

## 4

400 Bad Request, 164, 165, 166, 169, 170, 175, 218, 266, 278, 284, 360, 362, 364, 366, 368, 392, 435
401 Unauthorized, 265, 266
403 Forbidden, 164, 165, 169, 265, 266
404 Not Found, 80, 165, 218, 265, 266, 285, 360, 375, 376, 405
405 Method Not Allowed, 285
406 Not Acceptable, 198, 201
409 Conflict, 165
410 Gone, 192
415 Unsupported Media Type, 198

## 5

500 Internal Server Error, 164, 165, 265, 284
503 Service Unavailable, 370

## A

API-вызов, 236, 240, 241, 315, 316, 318, 319, 321, 334, 338, 340, 352, 358, 404
apihandyman.io, 32, 79, 80, 104, 115, 147
API Stylebook, 24
application/json, 135, 138, 197, 198, 303, 354, 355, 369
application/pdf, 197, 198
application/xml, 198, 199, 303, 369

## B

BFF, 343, 369, 379

## C

CRUD, 92, 101, 246, 247, 418

## G

GraphQL, 42, 104, 338, 339, 340, 343, 374, 376, 377, 378, 379
gRPC, 42, 104, 324, 338, 374, 375, 376, 377, 378, 379, 421

## H

HATEOAS, 209

## I

ISO 639, 199

ISO 3166, 199
ISO 4217, 191, 270, 306, 407, 430
ISO 7000, 191

## J

JSON Schema, 126, 127, 129, 130, 132, 133, 134, 136, 143, 187, 217, 389

## M

Markdown, 389, 394, 399, 403
MQTP, 380

## O

OpenAPI, 112, 113, 115, 120, 126, 134, 147, 153, 187, 194, 204, 211, 219, 220, 223, 231, 251, 253, 364, 384, 385, 387, 393, 394, 397, 398, 399, 402, 403, 406, 407, 408, 409, 410, 411, 412
OWASP, 244

## P

PUML 403

## R

ReDoc, 117, 118, 384, 385, 388, 395, 398, 402, 403, 407
REST API, 76, 77, 79, 80, 81, 82, 84, 92, 93, 101, 102, 103, 104, 106, 109, 112, 113, 115, 119, 143, 147, 157, 158, 164, 168, 170, 188, 192, 193, 197, 198, 200, 201, 203, 205, 206, 207, 208, 209, 210, 220, 231, 255, 266, 278, 281, 291, 295, 300, 303, 325, 338, 339, 373, 374, 375, 376, 377, 378, 379, 418, 421, 424

## S

SSE, 354, 355, 356, 357, 375, 378, 380
Swagger, 116, 117, 399

## W

WebDAV, 360, 361

## Y

YAML, 113, 115, 121, 123, 124, 129

## А

Адаптируемость, 195, 344

## З

Заголовки HTTP, 197, 294
Заголовок ETag, 324

## К

кеширование, 322, 324, 325, 326, 336, 340, 353, 375, 378, 379
кешируемость, 106
Кешируемость, 108
контекст, 42, 43, 44, 231, 233
критические изменения, 270, 271, 272, 273, 274, 277, 278, 279, 282, 284, 286, 289, 297, 301, 304, 309, 383, 415, 430

## Р

расширяемость, 302, 303, 304, 306, 307, 308

## У

управление версиями, 289, 290, 292, 294, 295, 296, 297, 298, 299, 300, 302, 309

## Ф

фильтрация, 202, 203
формат описания API, 109, 111, 112, 115, 117, 118, 119, 147, 251, 389, 435

## Ц

цели, 45, 51, 54, 55, 56, 58, 61, 62, 63, 66, 67, 68, 70, 71, 72, 75, 76, 80, 81, 82, 83, 84, 88, 92, 100, 101, 108, 109, 111, 112, 122, 126, 147, 149, 153, 162, 174

## Э

эффективность, 106, 311, 312, 325, 328, 330, 341, 342, 343, 344, 380

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.
При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. +7 (499) 782-38-89
Электронный адрес: **books@alians-kniga.ru**.

Арно Лоре

**Проектирование веб-API**

Главный редактор *Мовчан Д. А.*
*dmkpress@gmail.com*
Перевод *Беликов Д. А.*
Корректор *Абросимова Л. А.*
Верстка *Орлов И. Ю.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитуры «PT Serif», «PT Sans», «PT Mono» . Печать цифровая.
Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1.
Усл. печ. л. 35,75. Тираж 200 экз.
Веб-сайт издательства: **www.dmkpress.com**